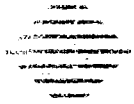
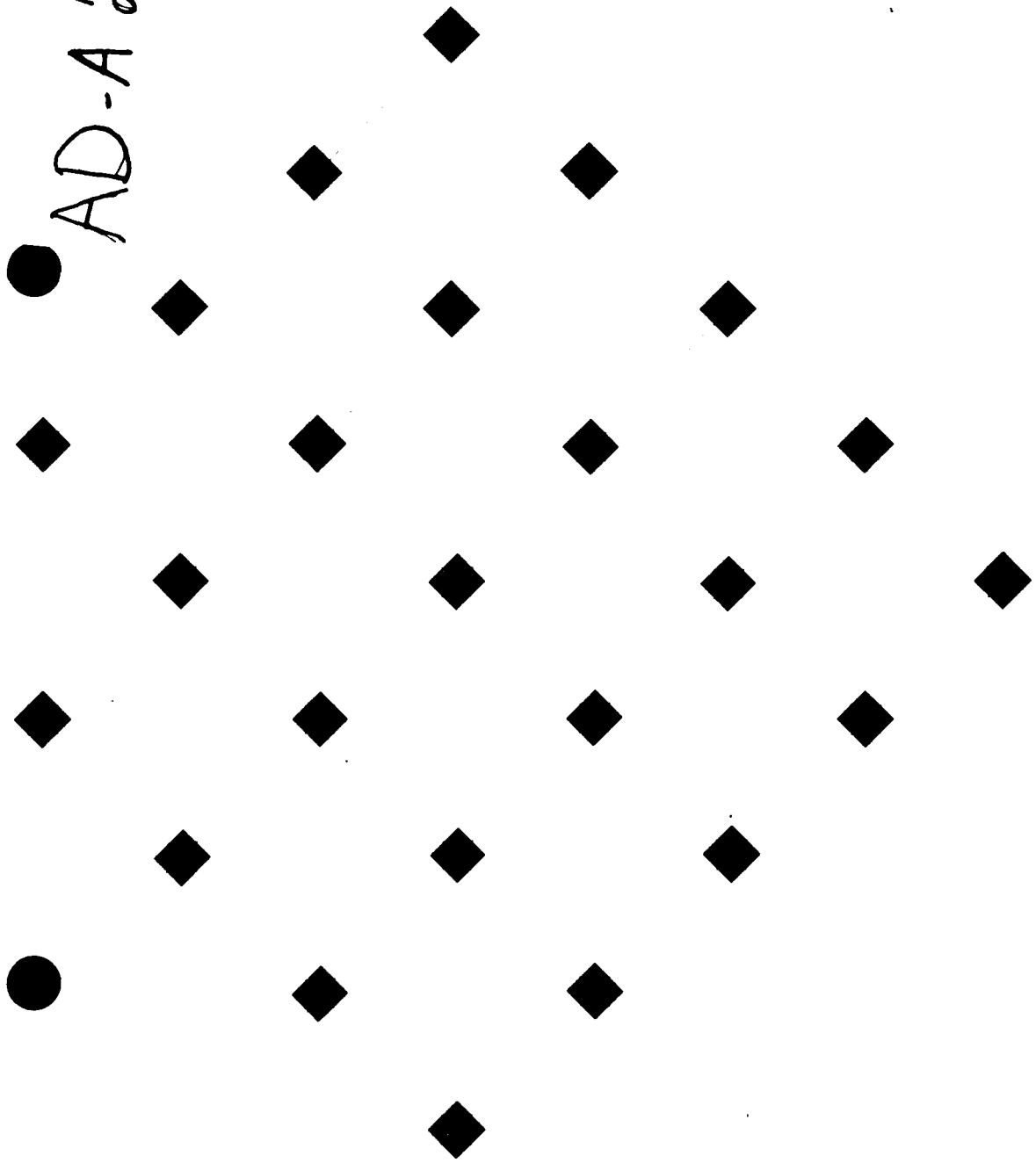


Software Engineering Institute

Software Specifications: A Framework

Curriculum Module SEI-CM-11-2.1



Software Specifications: A Framework

SEI Curriculum Module SEI-CM-11-2.1

January 1990

H. Dieter Rombach
University of Maryland

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.
Approved for public release. Distribution unlimited.

This document was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this document should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This document has been reviewed and is approved for publication.

FOR THE COMMANDER

JOHN S. HERMAN, Capt, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Software Specifications: A Framework

Acknowledgements

I would like to thank Norm Gibbs, Director of the SEI Education Program, who made sure I had the resources and encouragement to complete this work. Special thanks go to John Brackett, the author of the curriculum module *Software Requirements*, who reviewed earlier versions of this module and provided valuable feedback. I would also like to thank all the members of the Education Program, especially Gary Ford and Lionel Deimel for their helpful comments, Polly Bech for doing the graphical work, and Linda Pesante of Information Management for her editorial work.

Contents

Software Specifications: A Framework

Module Revision History

Version 2.1 (January 1990)	Minor revisions and corrections
Version 2.0 (December 1989)	Major revision Approved for publication
Version 1.0 (October 1987)	Draft for public review

Software Specifications: A Framework

Capsule Description

This curriculum module presents a framework for understanding software product and process specifications. An unusual approach has been chosen in order to address all aspects related to "specification" without confusing the many existing uses of the term. In this module, the term *specification* refers to any plan (or standard) according to which products of some type are constructed or processes of some type are performed, not to the products or processes themselves. In this sense, a specification is itself a product that describes how products of some type should look or how processes of some type should be performed. The framework includes:

- A reference software life-cycle model and terminology
- A characterization scheme for software product and process specifications
- Guidelines for using the characterization scheme to identify clearly certain life-cycle phases
- Guidelines for using the characterization scheme to select and evaluate specification techniques

Philosophy

Most SEI curriculum modules provide a structure for organizing a well-defined subject area (sometimes related to a life-cycle phase) and a guide for understanding the related literature. They are addressed to an educator audience, but contain material intended for presentation to students. This module has all these characteristics, but is atypical in the following ways:

- It is an *overview* module covering a broad subject area about which there is little consensus.
- It is intended to provide background for understanding other curriculum modules and is therefore addressed more to teachers than to students.
- It contains a good deal of original material, embodying an unusual approach to its subject matter.

The term "specification" is overloaded. It is used both informally and in the literature in a great variety of senses, and it is difficult to achieve a coherent understanding of the term that accounts adequately for this diversity. The resulting confusion may either be viewed as a simple terminology problem (*i.e.*: Which life-cycle products or processes should be referred to as "specifications"?) or as a more fundamental philosophical problem regarding the role of "specification" in the context of software development (*i.e.*: Can the notion of "specification" be restricted to certain life-cycle product and process types? Should only life-cycle products and processes, only their plans, or both objects and plans properly be called "specifications"?).

The Terminology Problem. According to [IEEE83], the term "software specification" refers either to a document or product that describes various characteristics of a software system or to the process of developing such a document or product. This general definition applies to a large variety of product and process types[1].

[1] In the study of software engineering, *individual* products or processes are of little interest. The term "type" is used here to denote the class of similar products or processes of which a particular one is an instantiation. Thus, for example, all Ada programs may be viewed as products of the same type (*i.e.*, Ada code products); all coding processes based on stepwise refinement that result in Ada programs may be viewed as processes of the same type (*i.e.*, stepwise-refinement–oriented Ada coding processes).

Many software development organizations have adapted this definition to their own technological and organizational characteristics and needs. The resulting terminologies are context-dependent and inconsistent regarding the use of the term "specification." Examples of inconsistencies between existing life-cycle terminologies include the following (see Figure 1, p. 20, middle column):

- The same term is used for product and process types (*e.g.*, "requirements definition," "system specification").
- The same term is used for different types of products (*e.g.*, "requirements specification," "functional specification").
- Different terms are used for the same type of product (*e.g.*, "requirements specification," "functional specification") or the same type of process (*e.g.*, "requirements analysis," "system specification").

Sometimes the same product may be referred to as "specification" or "implementation," depending on whether an executable specification language or a high-level implementation language is being used. Further, the same software characteristic may be addressed in one or more products, depending on the underlying life-cycle and project organization model. And processes may or may not be modeled explicitly, depending upon the perceived importance by the organization of "process."

The Philosophical Problem. Software development projects should be explicitly planned, executed, and evaluated. The project model depicted in Figure 2, p. 21, reflects these principles [Basili88]. It is definitely justifiable, based on the IEEE definition [IEEE83]—it is probably not an intended interpretation—to view both a number of life-cycle products and processes, as well as their plans resulting from the planning activity, as "specifications."

The purpose of planning is the production of "plans" —whether explicit or not—of what life-cycle products should look like and how life-cycle processes should be performed. Examples of such plans are, in the case of products, the ANSI/IEEE 830 standard on "software requirements specification" [IEEE84] and, in the case of processes, the DoD 2167A standard on "software development" [DoD88a] and the DoD 2168 standard on the "software quality assurance process" [DoD88b]. The purpose of execution is to perform processes and construct products according to their plans. The purpose of evaluation is to assess whether the plans were satisfactory and whether the life-cycle products and processes were constructed and performed in accordance with their plans.

A project model like the one depicted in Figure 2 enables us to address the sound selection and evaluation of software *specification techniques*, *i.e.*, models, languages, methods, and tools used to create life-cycle products or perform life-cycle processes according to their specifications. In practice, many major software development failures can be traced to the use of inappropriate (as well as inappropriate use of) techniques for describing software products and processes.

The Approach Taken Here. This module addresses the above problems by using a reference life-cycle terminology that avoids the term "specification" for any life-cycle product or process type. Instead, this module refers only to "plans" of product and process types as "specifications." Doing so eschews existing life-cycle terminologies in favor of one that facilitates consistency in the present exposition and allows the reader to reinterpret this module in terms of some other nomenclature he or she prefers, if necessary. In this module, then, a *software specification* is a product resulting from the planning process that prescribes how a product of some type should look or how a process of some type should be performed. This approach may seem unusual, but the author is convinced of its benefits.

Module Content. This curriculum module introduces the reference life-cycle model and terminology discussed above, builds a scheme for characterizing product and process specifications, uses this scheme to describe the process and product types related to certain life-cycle phases of the reference life-cycle model, and shows how such characterizations may be used to select and evaluate specification techniques.

Introduction of the reference life-cycle model and terminology depicted in Figure 1 (left and right columns, respectively) represents an attempt to overcome the confusion of terminology in the field. None of the product or process type names of the reference terminology uses the term "specification." However, cross references to some of the existing life-cycle terminologies are provided (Figure 1, middle column).

The scheme for characterizing product and process specifications is based on the following four dimensions:

1. Purpose and context (*i.e.*, what is the expected role of the specified product or process type?)
2. Content (*i.e.*, what aspects of the product or process type need to be described, and with what attributes?)

3. Representation format (*i.e.*, what models and languages should be used to represent the above content?)
4. Support (*i.e.*, what methods and tools should be used to support the creation of life-cycle products and processes according to the above representation format?)

The first two dimensions of the characterization scheme are used to identify three important phases in the context of the reference life-cycle model:

1. C-requirements (customer/user-oriented requirements)
2. D-requirements (developer-oriented requirements)
3. Design

These reference phases are discussed, using the framework, not because the author believes that they are more important than other phases, but because they are likely to correspond most closely to the reader's intuitive notion of "specification."

All four dimensions of the characterization scheme are used to select and evaluate specification techniques. Requirements for any specification technique are formulated in terms of the latter three dimensions of the characterization scheme, motivated by its project-specific purpose and context. Selection implies finding a specification technique that matches the stated requirements; evaluation implies comparing the actual effects of the chosen technique to the expected ones, as stated in the requirements.

Relation to Other Modules. It is helpful if the reader of this curriculum module is familiar with SEI curriculum modules *Models of Software Evolution: Life Cycle and Process* [Scacchi87] and *Technical Writing for Software Engineers* [Levine89].

Early life-cycle phases are often given less attention in the classroom than are later phases, such as design, coding, and testing, even though their importance is widely recognized. It is hoped that the insights into software specifications provided here will increase the understanding of teachers and allow these activities to be more widely taught.

This module provides material needed to understand software specifications and to apply that understanding to the characterization of specifications and to the selection and evaluation of specification techniques. No attempt is made to deal with system specifications or to provide detailed guidance about the production of particular life-cycle products. Instead, this module provides background for more narrowly focused curriculum modules, which utilize its terminology. Among these are *Software Requirements* [Brackett90], addressing C- and D-requirements, and *Introduction to Software Design* [Budgen89], dealing with design. Additional modules using the framework set forth here will follow. This module should be studied before reading any of these life-cycle–oriented curriculum modules.

Objectives

A person having studied the material covered in this curriculum module is expected to be able to do the following:

- Explain the nature of the confusion caused by the common uses of the term "specification."
- Apply the reference life-cycle model and relate its terminology to that of any of the commonly used models.
- Discuss C-requirements, D-requirements, and design within the framework presented in this module.
- Apply the characterization scheme to describe any process or product specification.
- Apply the characterization scheme to the selection of specification techniques.
- Apply the characterization scheme to the evaluation of specification techniques.

Prerequisite Knowledge

In order to understand this material, the student must understand the fundamentals of software engineering at the level of an introductory course and must have had practical software development experience as a member of a team.

Module Content

This module uses the terminology in [IEEE83] where possible. A glossary of significant terms follows the annotated outline.

Outline

I. Overview
  1. Conflicting Meanings of "Specification"
  2. Definition Used Here
  3. A Framework for Understanding Specifications
II. A Reference Software Life-Cycle Model and Terminology
III. A Characterization Scheme for Software Specifications
  1. Purpose and Context
    a. Product perspective
    b. Process perspective
    c. Use perspective
    d. People perspective
  2. Content
    a. Aspects
    b. Attributes
  3. Representation
    a. Models
    b. Languages
  4. Support
    a. Methods
    b. Tools
IV. A Characterization of Life-Cycle Phases
  1. C-Requirements
    a. Purpose and context
    b. Content
  2. D-Requirements
    a. Purpose and context
    b. Content
  3. Design
    a. Purpose and context
    b. Content
  4. Other Object Types
V. Guidelines for Selecting and Evaluating Specification Techniques
  1. Selection of Proper Specification Techniques
    a. Define specification requirements
    b. Chose specification techniques
  2. Evaluation of Specification Techniques
VI. Assessment of Current Maturity and Future Directions

Annotated Outline

I. Overview
  1. Conflicting Meanings of "Specification"

The term "software specification" is used inconsistently by the software community. Most of the time, it refers either to products created during the early phases of a software project, to the processes leading to those products, or to descriptions/characterizations of those types of products or processes.

  2. Definition Used Here

Although an argument can be made for referring to diverse types of products and processes by the term "specification," a compelling argument can also be made for restricting the term in order to avoid confusion. In this module, we will avoid completely use of the term for any of the usual life-cycle product or process types. Instead, we will define *software specification* as a plan or standard that provides a description/characterization of a software product or process type. This definition allows us to emphasize "good" software engineering, in that we focus on planning before execution.

A software specification, then, becomes a product resulting from the planning process. Execution of the "plan" results in the instantiation of a particular product or process. (See Figure 2, p. 21.) A *product specification* describes how products of some type should look; a *process specification* describes how processes of some type should be performed. In cases where planning is informal, implicit, or haphazard, specifications are not explicitly constructed.

Consider software design as an example. This might involve:

- The specification of the *input product type* (requirements product), including a formal syntax and semantics description for the requirements document, or a standard,

such as ANSI/IEEE-Std-830 on "software requirements specification" [IEEE84].

- The specification of the *output product type* (design product), including a formal syntax and semantics description for the design document.
- The specification of the *process type* (design process), including a guideline for the use of specific design techniques, such as Structured Design or object-oriented design.

As another example, consider software compilation, which might involve:

- The specification of the *input product type* (source code product), including the source language definition, a coding style handbook, and a language-oriented editor.
- The specification of the *output product type* (object code product), the object code definition.
- The specification of the *process type* (compilation process), the compiler tool itself.

Other examples of process specifications are the DoD 2167A standard on "software development" [DoD88a] and the DoD 2168 standard on "software quality assurance process" [DoD88b].

As a general rule, existing *specification techniques* —models, languages, methods, and tools used to instantiate specifications into life-cycle products or processes—are better suited (*e.g.*, are more formal) to the specification of (1) software product types, rather than process types, and (2) types used in later, rather than earlier, life-cycle phases.

3. A Framework for Understanding Specifications

This module presents a comprehensive framework for understanding software specifications and related issues. The framework includes:

- a reference life-cycle model and terminology,
- a characterization scheme for software product and process specifications,
- guidelines for using the characterization scheme to identify clearly certain life-cycle phases, and
- guidelines for using the characterization scheme to select and evaluate specification techniques.

The framework provides a tool for understanding the literature and provides background and context for other specification-related curriculum modules (*e.g.*, [Brackett90] and [Budgen89]).

Within the framework, we characterize any product or process specification by

- the purpose and context of the specified product or process type,
- the content of the type of product or process of interest,
- the representation format used to capture the content, and
- available support for the creation of the life-cycle products or execution of processes of the type of interest.

The characterization scheme can be used to

- Characterize the specification needs of a project.
- Characterize candidate specification techniques.
- Select the appropriate specification techniques by comparing the project specification needs with the characteristics of candidate specification techniques to find the best match.
- Evaluate specification techniques used by comparing observed characteristics to expected ones and, if necessary, suggest changes for future projects.

In this module, we will use the reference life-cycle model and characterization scheme to identify clearly several important life-cycle phases and to analyze these phases within our framework.

II. A Reference Software Life-Cycle Model and Terminology

Many different software life-cycle models exist (*e.g.*, waterfall [Royce70], iterative enhancement [Basili75], spiral [Boehm86], and prototyping [Boehm84]). They have in common certain types of products (*e.g.*, requirements, design, code). They differ substantially, however, in the types of processes used to build those products. From this observation, we may construct a reference life-cycle model that posits the existence of certain product types filling specific roles within a software development context but that makes no particular assumptions about the mechanisms by which products are actually built.

Such a reference life-cycle model is shown in the leftmost column of Figure 1, p. 20, where we assume the existence of the following product types (we do not distinguish between deliverable products and documents):

- *Software needs*, which are predominantly concerned with the questions: What demands exist? What needs should a proposed software product attempt to fulfill?
- *Customer/user-oriented software require-*

ments (C-requirements), which are predominantly concerned with the question: What functional and non-functional characteristics, from a customer's or user's point of view, must a product exhibit to meet those needs?

- *Developer-oriented software requirements (D-requirements)*, which are predominantly concerned with the question: What functional and non-functional characteristics, from a software developer's point of view, must a product exhibit to meet those needs?
- *Software design*, which is predominantly concerned with the question: How can a product be built to behave as described by the D-requirements?
- *Code*, which is predominantly concerned with the question: How is the product actually implemented on some machine using a particular technology?

The reference life-cycle terminology used in this curriculum module is depicted in the rightmost column of Figure 1. Whenever possible, we refer to processes and the resulting products of some type under the same name (*e.g.*, "design process" and "design product"). More detailed characterizations of the product and process types related to C-requirements, D-requirements, and design are contained in section IV.

Inconsistent terminologies are used in different industrial software development organizations and in the literature. Examples of commonly used terms are shown in the middle column of Figure 1. The reader may map his or her preferred or local terminology (and associated practice) to the reference terminology as required. Possible inconsistencies between the reader's terminology and the reference life-cycle terminology, along with resolutions enabling the application of our discussion to the reader's circumstances, include the following:

- The reader uses a different name than the reference model to refer to the same product or process type. Resolution is straightforward here, of course, as the reader can simply substitute one name for another. For example, the reader may prefer using the term "requirements definition" to refer to what we call "C-requirements product." The entire discussion of "C-requirements products" in this module applies to "requirements definitions," according to the reader's terminology.
- The reader identifies several types that collectively encompass one or more product or process types of the reference model, or vice versa, and a 1-1 mapping is not possible. In this situation, a more complex mapping is needed. For example, in the reader's terminology two product types, "behavioral specification" and "functional specification," may play the same role as our "D-requirements product." The entire discussion related to "D-requirements products" in this module applies to both "behavioral specifications" and "functional specifications."
- Due to the structural model chosen for the deliverable product, the reader deals with several instances of a product or process type of the reference model. In this case, multiple types may be distinguished with appropriate qualifiers and treated as instances of types described in this module. For example, if the product is structured into system, subsystems, and modules, the reader may identify a corresponding number of instances of types design product and design process.

III. A Characterization Scheme for Software Specifications

This section incorporates ideas from [Abbott86], [Sommerville89], [Firth87], and elsewhere. The scheme presented enables the characterization of any software product or process specification in terms of the purpose and context of the specified product or process type, the content of the specified type, the representation used, and the support for product creation or process execution.

1. Purpose and Context

Specifications describe all important characteristics of a particular software product or process type in some format. The desirable characteristics, as well as the appropriate format for representing them, are determined by the purpose and context of the type within the software development project. We have chosen to characterize purpose and context (in no particular order) from product, process, use, and people perspectives.

a. Product perspective

Product and process specifications are ultimately aimed at creating life-cycle products (*i.e.*, project deliverables) to satisfy the customer. Therefore, it is assumed that the choice of product and process specifications depends on the type of deliverables to be developed. We characterize product types by application and quality requirements.

(i) Application

The type of application has a deep impact on what product or process aspects (see section III.2.a) need to be specified. There are a number of possible classification schemes for software applications, for example:

- schemes based on control-flow char-

acteristics of the software system (sequential, concurrent, real-time)

- schemes based on the application (commercial, system, process control, scientific, embedded)

(ii) Quality requirements

The need to satisfy particular software quality requirements impacts both the aspects that need to be specified and their attributes (see sections III.2.a-b). For example, the need for maintainability may justify the explicit specification of the design rationale in a traceable form, so maintainers can trace changed requirements to affected design components.

An incomplete list of possible quality requirements includes:

- reliability
- correctness
- fault-tolerance
- maintainability
- portability
- user-friendliness
- availability

b. Process perspective

Specifications serve different purposes in different development process contexts. We characterize the process perspective in terms of the overall life-cycle model and its individual life-cycle phases.

(i) Life-cycle models

Different life-cycle models, reflecting different philosophies for creating software products, incorporate different product and process types [Scacchi87].

(1) Waterfall model

The waterfall model [Royce70] is based on the idea of producing product types at different levels of abstraction (requirements, system design, module designs, code) sequentially, followed by the integration of code in reverse order. Following this model in a project means transforming, in a linear fashion, the entire set of requirements into more and more concrete solutions. Attempting to feed lessons learned back into earlier stages results in (acceptable) deviations from the waterfall model. It must be remembered that the waterfall model is just a *model*, which is intended to stress the top-down principle for software development. In practice, there exist many exceptions to this sequential paradigm, reflecting the fact that errors are committed in the application of this principle.

(2) Iterative enhancement model

The iterative enhancement model [Basili75] is based on the idea of producing the same product types as for the waterfall model for only some of the requirements at a time. The idea is to allow for more effective learning-based feedback from each of these "mini-development" projects or to allow feasibility analysis of some critical requirements (by actually implementing them) before committing to the entire project. The product types used according to the iterative enhancement model might be the same used according to the waterfall model. However, the process types (or at least the instantiation patterns) are very different.

(3) Prototyping model

The prototyping model [Boehm84] is based on first concentrating on producing an operational software version for a limited set of the overall requirements. This limited set of requirements excludes part of the functional or non-functional overall requirements. Very often, crucial man-machine interface requirements or highly demanding performance requirements are the reason for prototyping. Prototyping is intended to help in the process of developing an acceptable C- or D-requirements product or to explore the technical feasibility of requirements and the associated risk. Prototyping is a way of learning "fast" about crucial project issues. The expectation is that this up-front investment pays off either by detecting early on that it is infeasible to continue the project or by creating an acceptable C- or D-requirements product that allows predictable and controllable software evolution. The goal is only to reuse the experience gained during the prototyping process and feed it back into creating better requirements, not necessarily to reuse any products created as part of the prototyping process. After acceptable requirements have been created, the regular software evolution process can follow any other life-cycle model (*e.g.*, waterfall).

(4) Spiral model

The spiral model [Boehm86] is based on a risk-driven approach to software evolution. Iterative development cycles are organized in a spiral manner, with inner cycles representing early analysis and prototyping,

and outer cycles representing the classic system life cycle. This technique is combined with risk analysis during each cycle. The model is intended to identify situations that might cause a development effort to fail or go over budget or schedule. The spiral technique incorporates ideas derived from the iterative enhancement model and the prototyping model.

(ii) Life-cycle products and processes

Product and process specifications are created for, used in, affected by, and modified during particular phases. These phases include, according to our life-cycle reference model:

- software needs
- C-requirements
- D-requirements
- software design
- code

Additional project phases may include:

- verification and validation [Collofello88]
- integration
- maintenance
- teaching and training

c. Use perspective

There exist a variety of different uses for specifications. We distinguish between uses for communication, creation, modification, verification and validation, and software quality assurance.

(i) Communication among people

Software projects include people. Specifications are aimed at supporting their communications regarding the important product and process characteristics and guidelines according to which products are created and modified, and processes are executed and changed. Specifications are a useful mechanism for teaching and training people what products should look like and how processes should be executed. Also, the existence of specifications allows project members to achieve reliable consensus about their roles by making explicit the project's purpose, context, and procedures.

(ii) Creation of products

Many software project tasks are aimed at creating, in a traceable way, instances of one product type from instances of another (*e.g.*, a design product from a D-requirements product). Explicit specifications for both product types and for the creating process (the design process, in this case) help guide and control the task. If all three specifications are completely formal (see [Berztiss87]), the desired product can be created automatically. In the best current practice, most product types are explicitly specified, whereas most process types are not. Further, downstream product types tend to be defined with greater formality that early-phase ones. The degree of formality and specificity in a process specification (or the lack thereof) is indicative of the possible degree of guidance and control. Process specifications can be used by people (*e.g.*, a designer uses a set of informal design guidelines) or by automated tools (*e.g.*, a compiler uses a formally specified procedure for transforming source code into object code).

(iii) Modification of products and processes

Software projects require the ability to react to changes. Changing product requirements during development or enhancement requests during maintenance typically requires modifications to existing products, with or without changing the underlying product specification. Changing project or environment characteristics (*e.g.*, addition of new personnel or introduction of new technology) may require modifications to existing processes and possibly to their underlying specifications. The existence of explicit product and process specifications permits the incorporation of changes in a systematic way.

(iv) Verification and validation products

The purpose of verification and validation (V&V) is to show that a life-cycle product of some type (*e.g.*, source code) is consistent with a life-cycle product of a different type (*e.g.*, design product) [Collofello88]. This kind of cross-checking between products is facilitated by the existence of explicit specifications.

(v) Assuring adherence to plans

Software quality assurance (SQA) is concerned with assuring that software development is carried out according to plan [Brown87]. Much of the concern of SQA, then, is with comparing software products and processes to their specifications. Examples are checking whether a design product is consistent with its specification or whether a review process was conducted according to established review guidelines.

d. People perspective

Specifications are created or used by audiences playing different project roles. Although some specifications are intended for consumption by

machines, people have to understand them in one way or another. Examples of different project audiences are listed below. (Some of the descriptions are adopted from [Firth87].)

(i) Customers

The audience that contracts for the software project and, in part, determines the C-requirements for the system.

(ii) End-users

The audience that will install, operate, use, and maintain the system after it is delivered, and that, in part, determines the C-requirements for the system.

(iii) Sub-contractors

The audience that performs development or maintenance activities contracted out by the primary development organization.

(iv) Requirements analysts

The audience that develops the C-requirements product in conjunction with the customers and end-users. Requirements analysts find a representation format appropriate to customer and end-user needs.

(v) Specification engineers

The audience that evolves the C-requirements product into the D-requirements product. The main objectives of specification engineers are to resolve ambiguities, remove inconsistencies, and represent the D-requirements in a format suitable for the development audiences. This often implies use of more formal representations for D-requirements than C-requirements.

(vi) Designers

The audience that describes how the software system is to be constructed to satisfy the corresponding D-requirements product. This involves making optimization decisions about the best way to proceed, given the constraints imposed in the D-requirements product. Examples of such constraints are performance requirements, resources available, and fault-tolerance capabilities. These constraints often influence the design as much as the required behavior of the system.

There are basically two types of design processes: (1) designing a system that consists of a set of communicating components and determining the functionality of the components, and (2) designing the algorithms and data structures encapsulated in those components.

(vii) Implementors

The audience that takes the component design products and develops the corresponding implementation products (code).

(viii) V&V personnel

The audience that checks whether life-cycle products are consistent with earlier life-cycle products.

(ix) SQA personnel

The audience that checks whether life-cycle products are created and life-cycle processes performed according to their specifications.

(x) Configuration management personnel

The audience that assures the integrity of software during and after development by initiating, evaluating, and controlling changes to the product [Tomayko87].

(xi) Maintenance personnel

The audience that keeps the software system operational and useful. Maintenance personnel perform corrective, perfective, and adaptive maintenance activities.

(xii) Managers

The audience concerned with filling leadership roles, controlling the budgets and schedules related to the project, ensuring that problems are recognized and resolved early, and dealing with personnel assignments and problems.

2. Content

We characterize a specification also by its content, that is, by the product or process aspects it addresses and by attributes to be possessed by the representation of those aspects.

According to this view, the roles of product and process specifications are not completely parallel. To begin with, mechanisms for specifying process types are much less developed than those for specifying product types. (More on this below.) More fundamentally, however, instantiation of a process specification produces action, whereas that of a product specification produces a static artifact, albeit one either capable of animation (*i.e.*, execution) or descriptive of another artifact with such a capability. Despite this difference, we will treat products and processes in parallel; examples will clarify the differences wherever applicable.

a. Aspects

Four important aspects that may be addressed in a specification are behavior, interface, flow, and

structure of the objects (products or processes) specified. To discuss these, we first introduce several definitions.

Dynamic characteristics of an object of any type relate to its use. Dynamic characteristics of a process can be captured during its execution (*e.g.*, the set of all design decisions made by a designer or historical data on the amount of time required for design on past projects). Dynamic characteristics of a product can be captured during its operational use by the customer/user or during its testing phase.

Static characteristics of an object of any type relate to its representation. Static characteristics of a process should be described in its specification (*e.g.*, the steps in a design process). Static aspects of a product are described in the product itself and in its specification (*e.g.*, data structures or algorithmic control structure of an Ada source code product).

Functional characteristics of an object of any type relate to its functional requirements. These can be identified by analyzing what services are provided by the object (*e.g.*, functions such as "store" and "retrieve" provided by a product; generation of a product of type "design" by a process).

Non-functional characteristics of an object of any type relate to its non-functional requirements. These can be identified by analyzing how services are provided by the object (*e.g.*, each of the above product functions must be provided in time less than *t*; the product of type "design" must be produced by the above process within a certain period of time and within a certain budget).

External characteristics of an object of any type relate to the black-box view of that object. External characteristics of an object can be identified without knowledge of its actual implementation (*e.g.*, a product provides certain interface functions or reacts to certain input stimuli in particular ways; a process consumes certain inputs and produces certain output products).

Internal characteristics of an object of any type relate to the white-box view of that object. Internal characteristics of an object are identified based on knowledge of its actual implementation (*e.g.*, a product contains a number of modules with certain bindings among them; a process consists of a number of subprocesses).

We now use these definitions to characterize, explain, and distinguish aspects of products and processes we may wish to address in specifications.

(i) Behavior (external, dynamic)

The externally observable response of a product or process to stimuli in actual use. Behavior may include externally observable states, outputs, or boundary conditions on the validity of inputs and states. We distinguish between functional and non-functional behavioral aspects.

(1) Functional behavior

This may include the response of a product to specific inputs or the requirement that a certain pre-condition of a process results (after execution) in a certain post-condition.

(2) Non-functional behavior

This may include response time of a product or the time allowed for completion of a process.

(ii) Interface (external, static)

The structure of the boundary between product or process and its environment. We distinguish between functional and non-functional interface aspects.

(1) Functional interface

This may include the set of functions provided by a product or the role a process plays in software development.

(2) Non-functional interface

This may include response-time constraints on a product or a description of the required synchronization points of a process with other processes.

(iii) Flow (internal, dynamic)

The internal dynamics of a product or process in actual use. This may include the flows of control, data, and information between structural units of the product or process. (The difference between control flow, data flow, and information flow is nicely explained in [Henry81].) In the case of parallel processes, we must also consider such aspects as synchronization. We distinguish among the following:

(1) Control flow between sub-products or sub-processes

(2) Data flow between sub-products or sub-processes

(3) Information flow between sub-products or sub-processes

(4) Synchronization between executing sub-products or sub-processes

(iv) Structure (internal, static)

The organization of a product or process into interacting parts. This includes the decomposition of the whole into components or the composition of the whole from basic units. Architectural, algorithmic, and data structures, as well as the internal interfaces between substructures, may be of interest. We distinguish among:

(1) Architectural structure of a product or process in terms of sub-products or sub-processes

(2) Interfaces between sub-products or sub-processes

(3) Algorithmic structure of a product or process

(4) Data structures used in a product or process

(5) Information structure across sub-products or sub-processes

b. Attributes

In general, each of the aspects in (a) can be represented in a variety of different forms. Purpose and context of the product or process type of interest require a suitable form of representation to exhibit certain attributes.

For example, if the aspect "data flow" of a design product needs to be validated, we may specify that its representation needs to exhibit the attributes "complete," "consistent," and "executable." If the "control flow" of a design process needs to be validated, we may specify that its representation needs to exhibit the attribute "executable."

(i) Correctness

Requirements are satisfied.

(ii) Completeness

All relevant information is captured.

(iii) Consistency

There are no internal or external contradictions.

(iv) Feasibility

Requirements are satisfied within the constraints imposed by the software evolution context.

(v) Non-ambiguity

Alternative interpretations are not possible.

(vi) Clarity

The meaning of the representation is easily understood and communicated.

(vii) Preciseness

The meaning is exact.

(viii) Formality

Formal syntax and semantics are used. Various degrees of formality are possible. Mathematical formalism is the subject of [Berztiss87], [Bjørner82], [IWSSD82], [IWSSD84], [IWSSD85], and [IWSSD87].

(ix) Abstractness

The description is at a particular level of abstraction. D-requirements are more abstract (removed from the details of the eventual implementation) than code.

(x) Structuredness (or modularity)

The description shows systematic structure. Lessons learned regarding the production of readable code by applying the concepts of modularization and minimizing interfaces between modules should be applied to specifications of all types of products and processes.

(xi) Traceability

One is able to relate information items of corresponding product or process types. For example, a C-requirements product is much more helpful in the context of maintenance if it is possible to trace changes made to the D-requirements to certain components described in the architectural design product.

(xii) Modifiability

Changes can be made easily whenever necessary (*e.g.*, during maintenance).

(xiii) Executability

The attribute of being automatically executable on some machine. This characteristic allows for validating the dynamic and behavioral characteristics; the executability of more abstract products (*e.g.*, D-requirements) underlies the quick-feedback idea of prototyping.

(xiv) Verifiability

Techniques (possibly formal) can be used to check for consistency with requirements.

3. Representation

Certain software aspects (see III.2.a) need to be represented so they exhibit desired attributes (see III.2.b). The representation format chosen is based on models and languages. Models allow the formulation of aspects of interest. Languages allow the well-defined reflection of those models in a form that exhibits the desired attributes. We make the

distinction between models and languages to express the different formal representational capabilities. In practice, however, it is not always easy to distinguish between models and languages.

Our discussion may seem to be biased toward products, rather than processes. In fact, despite the recognized need for representing "process," most people use traditional product languages for the purpose. It is currently a burning research issue to identify appropriate mechanisms for process representations. (*E.g.*, see the annual proceedings of the International Software Process Workshop, which are usually published as special issues of ACM SIGSOFT's *Software Engineering Notes.*)

a. Models

Specification models allow the formulation of and reasoning about certain aspects of interest.

An incomplete list of examples includes:

- functional models
  - input-output models [Ross77]
  - algebraic models [Guttag78]
  - axiomatic models [Hoare69]
- finite state models [Parnas72]
- statecharts [Harel88a]
- stimulus-response models [Alford77]
- Petri net models [Peterson77, Bruno86, Peterson81]
- control flow models
- constraint models
- module interconnection models [DeRemer76]
- data structure models
- information flow models
- information structure models
- requirements net models [Alford77]
- data flow models [Babb85]
- entity-relationship models
- relational models [Teichrow77]

b. Languages

Specification languages allow the presentation of specifications in a well-defined fashion [Balzer81]. It is impossible to give a complete list of such languages; there are just too many. Most of them allow the representation of more than one aspect of the thing specified. For example, an implementation language such as Ada allows representation based on control flow, data flow, and data structure models. Instead, we provide a characterization of existing languages based on whether they are formal or semi-formal or informal, whether they are textual or graphical, and the language paradigm on which they are based.

We distinguish between formal, semi-formal, and informal languages:

- *Formal* languages are based on formal syntax and semantics [Berztiss87].
- *Semi-formal* languages are based on some formal syntax and are usually graphically oriented.
- *Informal* languages are usually based on natural language.

Most of the product (and process) specification languages used in practice are semi-formal languages, combining formal and informal elements. Most are based on a conceptual specification model, a specific representation, or a development approach.

We distinguish between

- *tabular*,
- *textual*, and
- *graphical*

representation languages.

We also distinguish between different language paradigms. Some important examples are:

- imperative
- declarative
- constraint oriented
- data-flow oriented

The reader interested in different language paradigms is referred to any classical programming language book.

4. Support

In practice, it is necessary to have effective support for creating specifications, as well as for using them during project execution. Most existing support actually addresses the instantiation of products according to product specifications. We distinguish between methods that provide operational guidelines based on some models and/or languages, and the automation of those guidelines using computers. There exists a *m*-to-*n* relationship between methods and tools. One method can be supported by an integrated set of tools, a single tool, or several tools alternatively. Correspondingly, a tool may support part of a method, an entire method, or several integrated methods.

a. Methods

Popular examples include:

- SREM [Alford77]
- Jackson Methodology [Cameron89, Sutcliffe88, Cohen86]

- SADT [Ross77]
- PSL [Teichrow77]
- Structured Analysis [DeMarco79, Yourdon89]
- Real-time specification methods [Rzepka85, Hatley87]

These methods are discussed in detail in relevant SEI curriculum modules (*e.g.*, SA, SADT, and SREM in [Brackett90]).

b. Tools

Popular examples include:

- PSA [Teichrow77]
- REVS [Alford77]
- compilers and runtime environments for all kinds of languages [Goldsack85]
- Statemate [Harel88b]

These tools will be discussed in detail in the appropriate curriculum modules.

IV. A Characterization of Life-Cycle Phases

In this section, the characterization scheme of section III is used to define some of the phases within the reference life-cycle model of section II. We will provide definitions of C-requirements, D-requirements, and design based on the purpose/context dimension (section III.1) and the content dimension (section III.2.a) of the characterization scheme. Figures 3a and 3b allow for the graphical representation of such definitions. First, we characterize the purpose/context of a specification within some software evolution process (vertical axis in Figure 3a). Second, we derive the aspects that need to be described based on purpose/context (horizontal axis in Figure 3a). Third, we define desirable attributes for each aspect (vertical axis in Figure 3b), considering also purpose/context. Marked matrix elements in Figures 3a and 3b provide a graphical representation of the scope of the corresponding life-cycle phases of interest.

These definitions help us define particular software development activities and serve to delineate the bounds of related curriculum modules, such as those on requirements analysis [Brackett90] and design [Budgen89].

1. C-Requirements

C-requirements are predominantly concerned with answering the question *what functional and non-functional characteristics, from a customer's/user's point of view, must a software product exhibit?* This section characterizes products of type C-requirements, using the characterization scheme introduced in section III (partly reflected in figure 4). Processes of type C-requirements are is treated in [Brackett90].

a. Purpose and context

The purpose and context of products of type C-requirements can be characterized as follows:

- **Product Perspective:** For our purposes here, we generalize across all possible application domains and quality requirements.
- **Process Perspective:** For our purposes here, we generalize across all possible life-cycle models. We are interested in product and process types related to overall system requirements and their validation.
- **Use Perspective:** C-requirements serve as a basis for communication with the customer and end-user. They define, in a contractual sense, what functions a software system must fulfill. In addition, they serve as input product for the subsequent creation of the D-requirements, as reference document for acceptance testing (V&V), and as the potential starting point for maintenance activities (especially in the case of perfective maintenance). The C-requirements product is derived from software needs; created during the C-requirements process; used during design, verification and validation, and maintenance activities; and modified throughout the entire lifetime of the corresponding software system.
- **People Perspective:** C-requirements are used by customers and end-users, requirements analysts, specification engineers, verification and validation people, quality assurance personnel, maintenance personnel, and managers.

b. Content

The content of C-requirements can be characterized as follows:

- **Aspects:** C-requirements address the aspects *behavior* and *interface*, insofar as they are important to establish a contractual relationship with the customer and user. Sometimes even structural aspects (*i.e.*, design constraints) have to be addressed if they are essential to product creation (*e.g.*, in the case of a specific technical process that needs to be controlled).

  C-requirements can suffer from over-specification, as well as under-specification. Of course, it is desirable to describe all aspects that are of interest to the customer and user as com-

pletely as possible. On the other hand, unnecessarily included items can restrict the subsequent development choices needlessly.

Abbott [Abbott86] provides a non-exhaustive list of C-requirements issues:

- why the user wants the system
- how the user intends to use the system
- what other systems and procedures will interface with the planned system
- what expertise the people have who will actually operate the system
- what information the system must be able to handle
- whether there are any legal constraints (*e.g.*, record retention requirements)
- whether the system must enforce any integrity constraints (*e.g.*, access limitations)
- what data processing functions the system should perform for the user.

Optional issues include:

- on what hardware must the planned system operate
- in what programming language must the system be written
- on what operating system must the system be installed
- what expected load must the system be able to handle (*e.g.*, in transactions per hour)
- what response time is needed from the system
- what enhancements must be expected for the system after initial use
- what design qualities are expected of the system
- what auditing processes must be performed
- what physical constraints exist for the system (*e.g.*, need for air conditioning because of location)
- what peripheral devices must be used

- **Attributes:** The desirable attributes of C-requirements cannot be characterized easily without knowing the life-cycle context and the application context. Each of the attributes in section III.2.b might be of importance under certain circumstances. However, the most desirable attributes of C-requirements are completeness (at least from the customer's perspective), consistency, and clarity. In addition, depending on the need for changes, it may be desirable for the product to be structured, traceable, and formal.

2. D-Requirements

The purpose of D-requirements is to answer the question *what functions, from a developer's point of view, must a software system fulfill?* This section characterizes products of type D-requirements, using the characterization scheme introduced in section III (partly reflected in Figure 5). Processes of type D-requirements are treated in [Brackett90].

a. Purpose and context

The purpose and context of products of type D-requirements can be characterized as follows:

- **Product Perspective:** For our purposes here, we generalize across all possible application domains and quality requirements.
- **Process Perspective:** For our purposes here, we generalize across all possible life-cycle models. We are interested in product and process types related to overall system requirements and their validation.
- **Use Perspective:** D-requirements define, for the software developer, the functional and non-functional characteristics the product under development must fulfill. Therefore, D-requirements serve as a basis for communication with the developer. In addition, they serve as input product for the subsequent creation of the software design, as reference document for the integration and system testing (V&V), and as the potential starting point for maintenance activities (especially in the case of adaptive maintenance). The D-requirements product is evolved from the C-requirements; created during the D-requirements process; used during design, verification and validation, integration, and maintenance activities; and updated throughout the entire lifetime of the corresponding software product.
- **People Perspective:** D-requirements are used by sub-contractors, specification engineers, designers, verification and validation people, quality assurance personnel, maintenance personnel, and managers.

b. Content

The content of D-requirements can be characterized as follows:

- **Aspects:** D-requirements address the aspects *behavior*, *interface*, and *structure*, insofar as they are important to the developers. Due to the difference in audience, D-requirements typically are specified in a different format from that used for C-requirements. Often more formal languages are used (*e.g.*, state-machine languages) than for C-requirements (*e.g.*, SADT).
- D-requirements, too, can suffer from over-specification, as well as under-specification. Subsequent development choices should not needlessly be restricted.
- **Attributes:** The desirable attributes of D-requirements cannot be characterized easily without knowing the life-cycle and the application contexts. Each of the attributes in section III.2.b might be of importance under certain circumstances. However, the most desirable attributes of D-requirements are completeness (at least from the developer's perspective), consistency, formality, traceability (from the C-requirements, to the design), and structuredness. Traceability from the C-requirements specification can be easily achieved if the D-requirements specification evolves from the C-requirements specification, rather than being a completely new product.

3. Design

The purpose of a design is to answer the question *how can a system be built to behave as described in its related D-requirements?* This section characterizes the design phase, using the characterization scheme introduced in section III (partly reflected in Figure 6). Processes of type design are treated in [Budgen89].

a. Purpose and context

The purpose and context of products of type design can be characterized as follows:

- **Product Perspective:** For our purposes here, we generalize across all possible application domains and quality requirements.
- **Process Perspective:** For our purposes here, we generalize across all possible life-cycle models. We are interested in product and process types related to overall system requirements and their validation.
- **Use Perspective:** Design products serve as a basis for communication with the subsystem or module designer or implementor. In addition, they serve as input product for the subsequent creation of the subsystem or module design or implementation, as reference document for the module or subsystem integration testing (V&V), and as the potential starting point for local maintenance activities. A design product is derived from its related D-requirements product; created as the result of a design process; used during design, implementation, verification and validation, integration, and maintenance activities; and modified throughout the entire lifetime of the corresponding software system.
- **People Perspective:** Designs are used by designers, implementors, verification and validation people, quality assurance personnel, maintenance personnel, and managers. They define the overall structure of the software system to be built. They define subsystems or modules, their functional requirements, and interfaces between them. The functional requirements serve as the input for the subsystem/module design activities, as do the D-requirements for the overall system design phase.

b. Content

The content of a design product can be characterized as follows:

- **Aspects:** Designs address the aspects *flow* and *structure*, insofar as they are important for further development. Which specific software aspects need to be specified predominantly depends on the project and application type. In the case of information systems, the data structure might be dominant; for embedded systems, control flow, interfaces, and synchronization might be dominant. Practical constraints during design may include (1) the consideration of ties between the software system under development and its anticipated target environment and (2) the awareness of compatibility with the chosen implementation language and hardware.
- **Attributes:** The desirable attributes of designs cannot be characterized easily without knowing the life-cycle and application context. Each of the attributes

in section III.2.b may be of importance under certain circumstances. However, the most desirable attributes of designs are completeness (at least from the implementor's perspective), consistency, formality, traceability (from the D-requirements, to lower-level designs or code), clarity, and structuredness.

4. Other Object Types

There are many other software objects for which sound specifications are needed. Examples are:

- management processes (*e.g.*, monitoring schedule adherence)
- management products (*e.g.*, schedules)
- other life-cycle processes (*e.g.*, testing)
- analysis processes (*e.g.*, measurement)

It is important to understand all aspects of the software life-cycle. The first step to better understanding is the ability to specify all aspects. The more formally a process or product type can be specified, the better it can be communicated, taught, executed, and improved. Broadening our view of life-cycle objects that need to be specified from just the conventional products (including documents) to all types of products and processes involved in software evolution is the objective of this section.

Individual software development organizations establish their own specification standards. Most of these standards are not well documented. The two major sources of standards are the Department of Defense and ANSI/IEEE. Examples of standards from those sources are:

- DoD Std 2167A on "Software Development" [DoD88a]
- DoD Std 2168 on "Software Quality Assurance" [DoD88b]
- ANSI/IEEE Std 830 on "Software Requirements Specification" [IEEE84]

V. Guidelines for Selecting and Evaluating Specification Techniques

One important application of the characterization scheme of section III is its use in selecting and evaluating specification techniques. Although this is an important topic, we can deal with it only briefly here.

1. Selection of Proper Specification Techniques

For selecting an appropriate specification technique, the framework should be applied as follows:

a. Define specification requirements

- Explicitly define purpose/context (III.1) and aspects (III.2.a) of the specification type of interest by using the matrix depicted in Figure 3a. The selection of methods and tools only makes sense in the context of a specific project or project type.
- Derive, for each specification aspect of interest, the appropriate attributes (II.2.b), using the matrix depicted in Figure 3b.

b. Chose specification techniques

- Select models and languages (III.3) that best match the derived aspect-attribute matrix. Obviously, this selection would be most efficient if we had definitions of a number of candidate models and languages in the form of the matrix in Figure 3b.
- Select methods (II.5.a) and tools (II.5.b) that best support use of the selected models and languages.

2. Evaluation of Specification Techniques

The evaluation of techniques needs to be done with respect to some goal [Basili88]. The characterization of a technique according to our framework has two advantages in this context: (1) it provides valuable input as to what evaluation goals might be of interest (*e.g.*, quality requirements [III.1.a.ii]), and (2) it provides a basis for relating negative or unsatisfactory observations regarding the effects of a technique to particular characteristics or to actual use of the technique (*e.g.*, a C-requirement technique may be ineffective because it is too formal for the customer to understand).

We can think of two kinds of evaluations: (1) evaluating whether a chosen technique actually possesses the characteristics promised by its creator or expected by us or (2) evaluating whether a chosen technique achieves the expected impact on software quality or productivity.

The first type of evaluation is relatively easy. The evaluation goal is implicitly defined by the original characterization of the technique on which its selection was based (see IV.1). We can develop a second characterization during the use of the technique in a real project, reflecting our actual experience. This actual characterization can then be compared with the original characterization.

The second type of evaluation requires more planning. Evaluation goals should identify the perspective (*i.e.*, the audience for this evaluation), which can be derived from the people dimension of our framework, as well as a characterization of the environment (the life-cycle model that was used and the application type), which can be derived from the process and application context dimensions of our

framework. Perhaps the hardest part of the evaluation process for specification techniques is formulating recommendations about what should be improved: train people better, choose better techniques, make sure that techniques are more thoroughly applied, or apply different life-cycle models or management structure. Defining expectations for the use of a technique based upon our framework and selecting it according to the procedure presented in section IV.1. allows comparison of expectation to reality, thus providing a more objective basis on which to improve existing techniques or select better-suited ones than is otherwise available.

It is important to recognize that evaluation, although potentially time-consuming and expensive, is necessary to guarantee improvement in the way we select and use specification techniques.

VI. Assessment of Current Maturity and Future Directions

Many people have a limited view of what software life-cycle objects are subject to specification and how they should be specified. Commonly held beliefs include:

- Only product types, not process types, need be specified.
- Product types in later phases of the life cycle should be specified more formally.
- Specifications are mostly used for purposes of communication and validation.

These attitudes provide fertile ground for change. Future developments are likely to include:

- The broadening of the notion of specification to all product and process types in the context of software evolution.
- The development of specific process specification languages.
- The introduction of greater formality of specification.
- The generation of custom-tailored environment components (*e.g.*, database schemes) from specifications of the software processes to be supported.

This prediction is motivated by the many needs of the software community, all ultimately aimed at improving productivity and quality of software evolution and its resulting products:

- Better understanding of the software evolution process itself.
- Better control of process executions.
- Better traceability and predictability of the impact of decisions made early in the project.
- Better basis for reuse.
- Better basis for constructing automated environments that actually support some predefined set of processes.
- A basis for employing generator technology for building environment components from process specifications.

Glossary

The following terminology is used throughout the module, except in the abstracts found in the bibliography.

process

Each activity or action that consumes (or is intended to consume) input products and/or produces (or is intended to produce) output products, *e.g.*, the overall software life cycle, each life-cycle activity (such as designing or testing), or even each action of the compilation process. *Process* is used in a very general sense.

process type

A class of processes with common characteristics. For example, all development processes executed according to some standard *X* are said to be of type *X*.

product

Each document or artifact created during (or for) a project is a product, independent of whether or not it is designated for delivery to the customer (*e.g.*, design document, code, measurement data, project plan). This is a broader definition than that of the IEEE ("[a] software entity designated for delivery to a user") [IEEE83].

product type

A class of products with common characteristics. For example, all requirements products created according to some standard *X* are said to be of type *X*.

project execution stage

The project activities concerned with performing the project according to the plans (specifications) produced in the preceding planning stage. (See *software project model.*)

project feedback stage

The project activities concerned with monitoring the effectiveness of the specifications used dur-

ing the execution stage, evaluating those results after execution, and feeding them into the planning stages of future projects. (See *software project model.*)

project planning stage

The project activities preceding the actual execution stage of a project. This stage is concerned with creating specifications of all relevant product and process types. This includes all products, whether deliverables or not, and processes for management, construction, control, and analysis. (See *software project model.*)

requirement

Any function, constraint, or other property that must be provided, met, or satisfied to fill the needs of the system's intended user(s) [Abbott86].

software development

The process of translating customer/user needs into a system for operational use [IEEE83].

software evolution

The process of software development and maintenance.

software maintenance

The process of modifying a product after delivery to correct faults (*corrective maintenance*), to improve performance or other attributes (*perfective maintenance*), or to adapt the product to a changed environment (*adaptive maintenance*).

software project model

The software project model underlying this curriculum module is based on (1) planning, (2) execution, and (3) evaluation-based feedback stages. Conventional life-cycle models describe the execution part. Specifications are created during planning, used to control the performance of processes and the creation of products during execution, and evaluated after execution during the feedback stage.

specification

A plan or standard that provides a description/characterization of a software product or process type. A specification is itself a product resulting from the planning process. A process specification describes how processes of some type should be performed; a product specification describes how products of some type should look. Having process and product specifications available allows us to instantiate individual processes and products from such specifications during project execution. It should be clear that the term *specification* refers to the description of a product or process type, not to the individual product or process.

Among its definitions for *specification*, [Webster87] gives:

1. The act or process of specifying.
2. A detailed precise presentation of something or of a plan or proposal for something ... a statement of legal particulars (as of charges or of contract terms).

According to [IEEE83], specification in the context of software engineering is:

1. A document that prescribes, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or system component.
2. The process of developing a specification.

Figures

Figure 1. Reference life-cycle model and terminology.

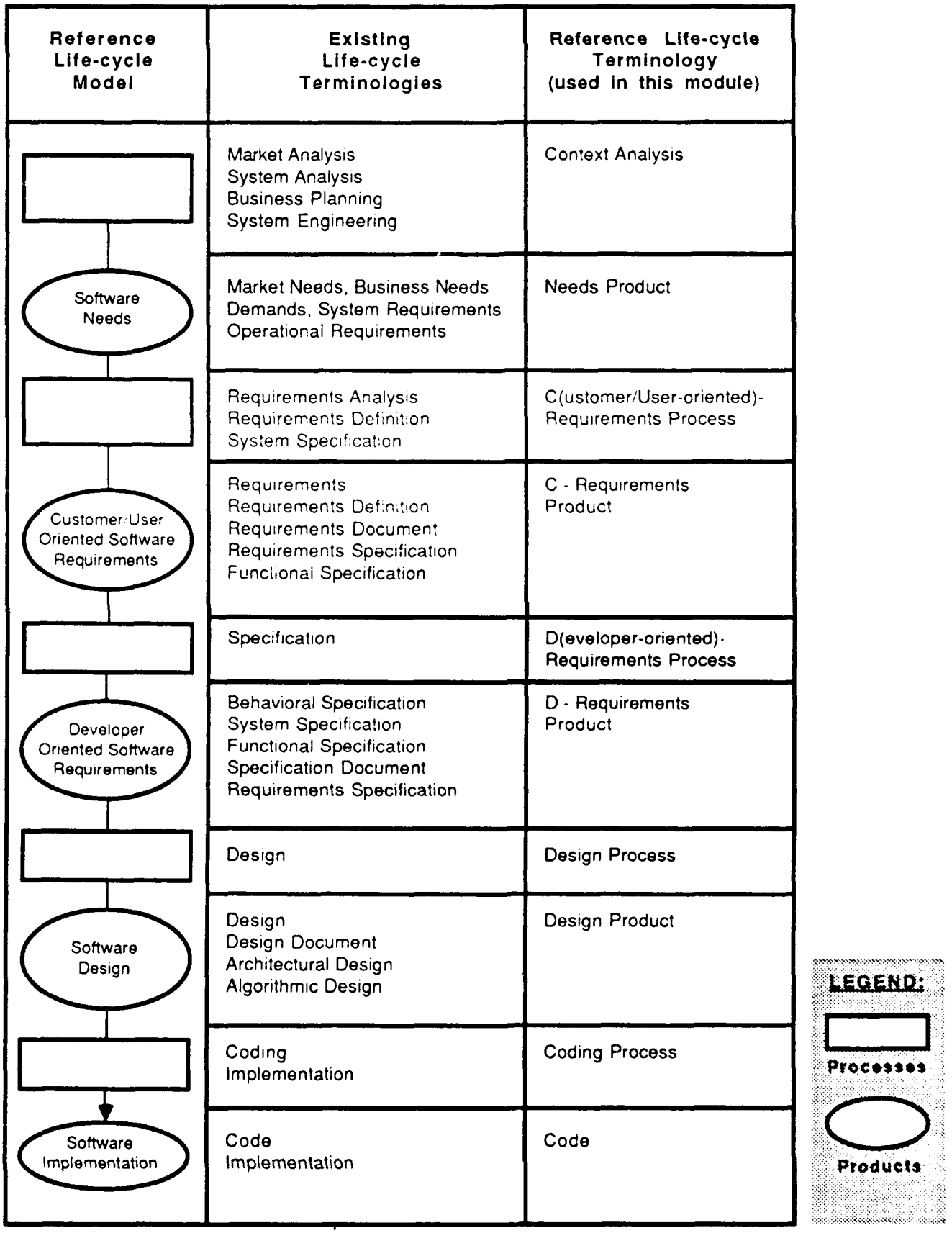


Figure 2. Planning/Execution/Feedback-Based Project Model.

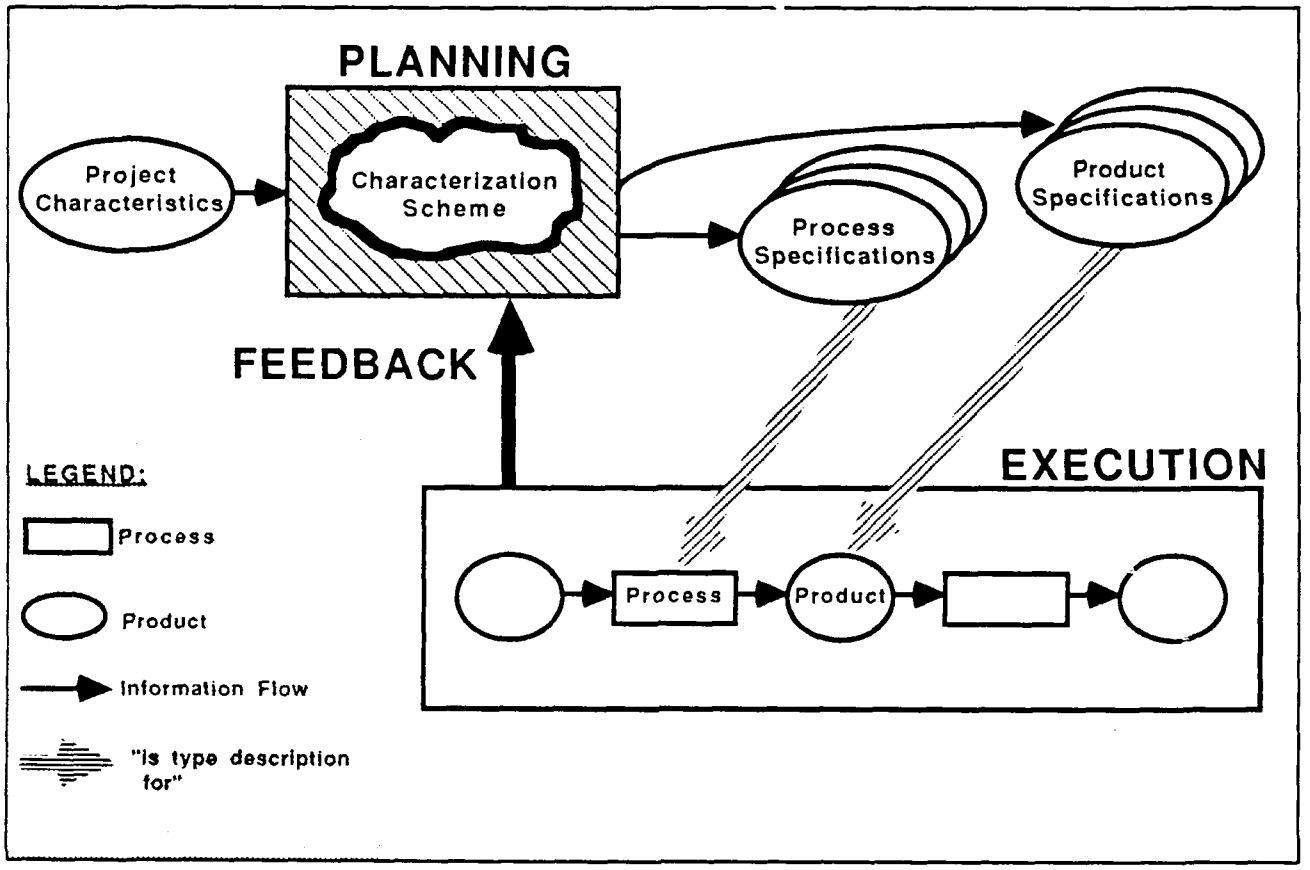


Figure 3a. Purpose & Context vs. Content (Aspect).

PURPOSE & CONTEXT		ASPECT: III. 2. a.												
		Behavior		Interface		Flow				Structure				
		Functional	Non-Func.	Functional	Non-Func.	Cont.Flow	Data Flow	Inf. Flow	Synchr.	Architect.	Interface	Algor. Str.	Data Str.	Inf. Str.
Product III. 1. a	Application Type 1:													
	• Sequential													
	• Concurrent													
	• Real-time													
	Application Type 2:													
	• Commercial													
	• Systems													
	• Process control													
	• Scientific													
	• Embedded													
	⋮													
	Quality Requirements:													
	• Reliability													
	• Correctness													
	• Fault-tolerance													
	• Maintainability													
	• Portability													
	⋮													
Process III. 1. b.	Life-Cycle Models													
	• Waterfall													
	• Iterative enh.													
	• Prototyping													
	• Spiral													
	⋮													
	Life-Cycle Phases													
	• Requirements													
	• Design													
	• V & V													
	• Integration													
	• Maintenance													
	• Teaching													
	⋮													
Uses III 1. c	• Communication													
	• Creation													
	• Modification													
	• V & V													
	• Assurance													
People III 1 d.	• Customer													
	• End-user													
	• Sub-contractor													
	• Req. analyst													
	• Spec. engineer													
	• Designer													
	• Implementor													
	• V & V pers.													
	• QA pers.													
	• Conf. man. pers.													
	• Maintenance pers.													
	• Manager													

Figure 3b. Content (Attributes) vs. Content (Aspects).

CONTENT (ASPECTS) - III. 2. a.														
		Behavior		Interface		Flow				Structure				
		Functional	Non-Functional	Functional	Non-Functional	Control Flow	Data Flow	Information Flow	Synchr..	Architectural Str.	Interface Str.	Algorithm. Str.	Data Structure	Info. Structure
CONTENT (ATTRIBUTES) -- III. 2. b.	Correctness													
	Completeness													
	Consistency													
	Feasability													
	Non-Ambiguity													
	Clarity													
	Preciseness													
	Formality													
	Abstractness													
	Structuredness (Modularity)													
	Traceability													
	Modifiability													
	Executability													
	Verifiability													

Figure 4. C-Requirements Characterization.

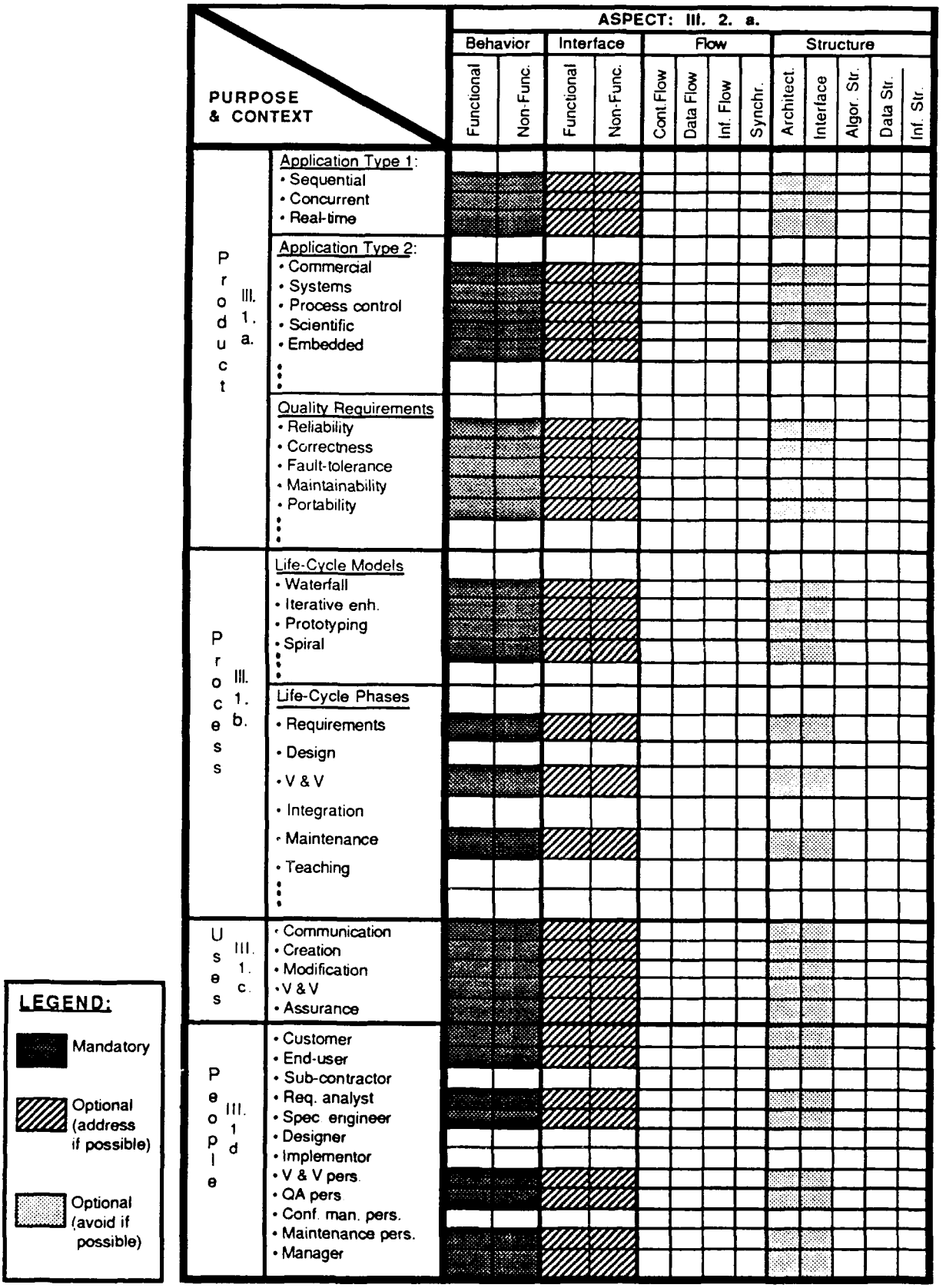


Figure 5. D-Requirements Characterization.

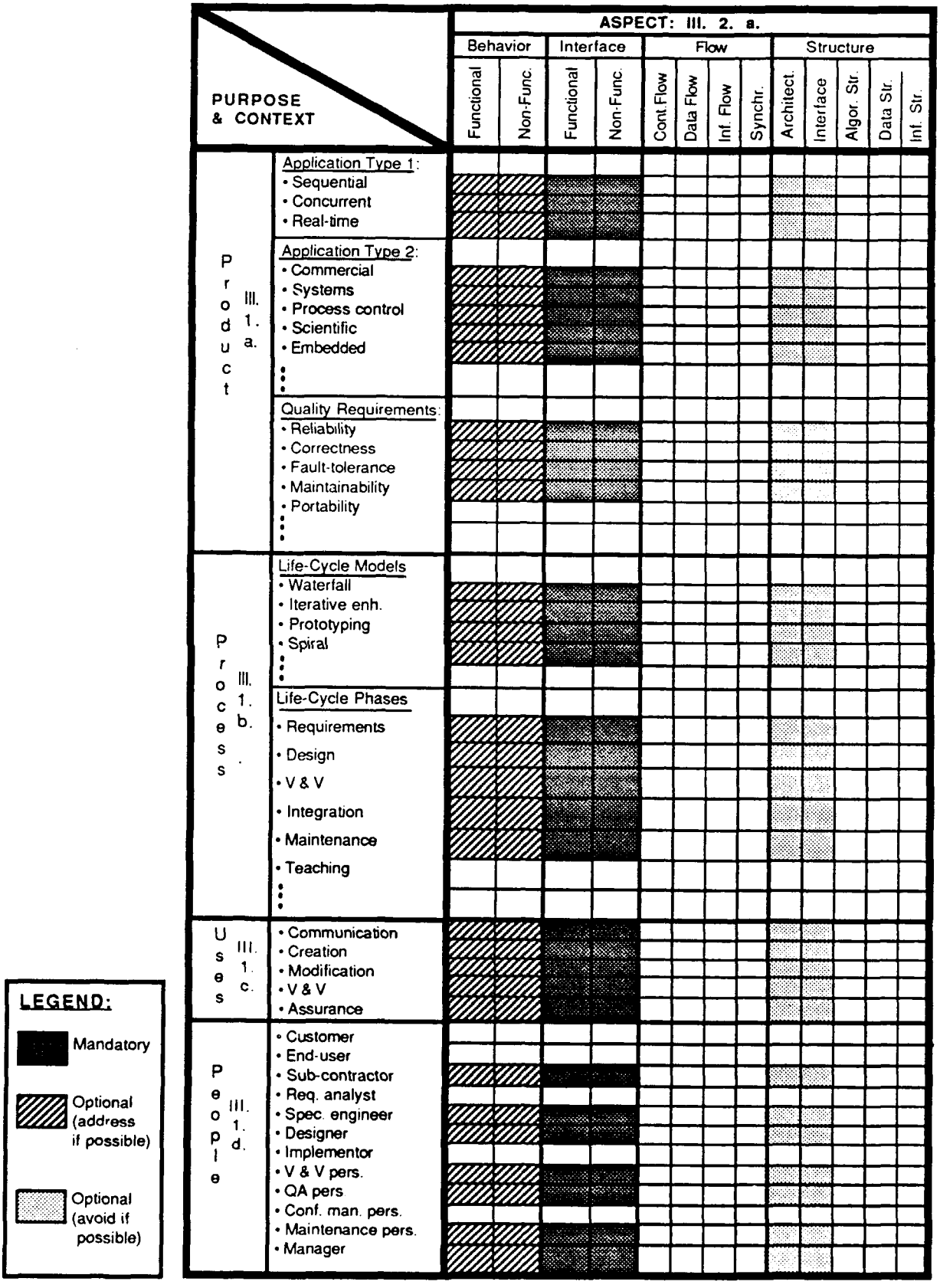
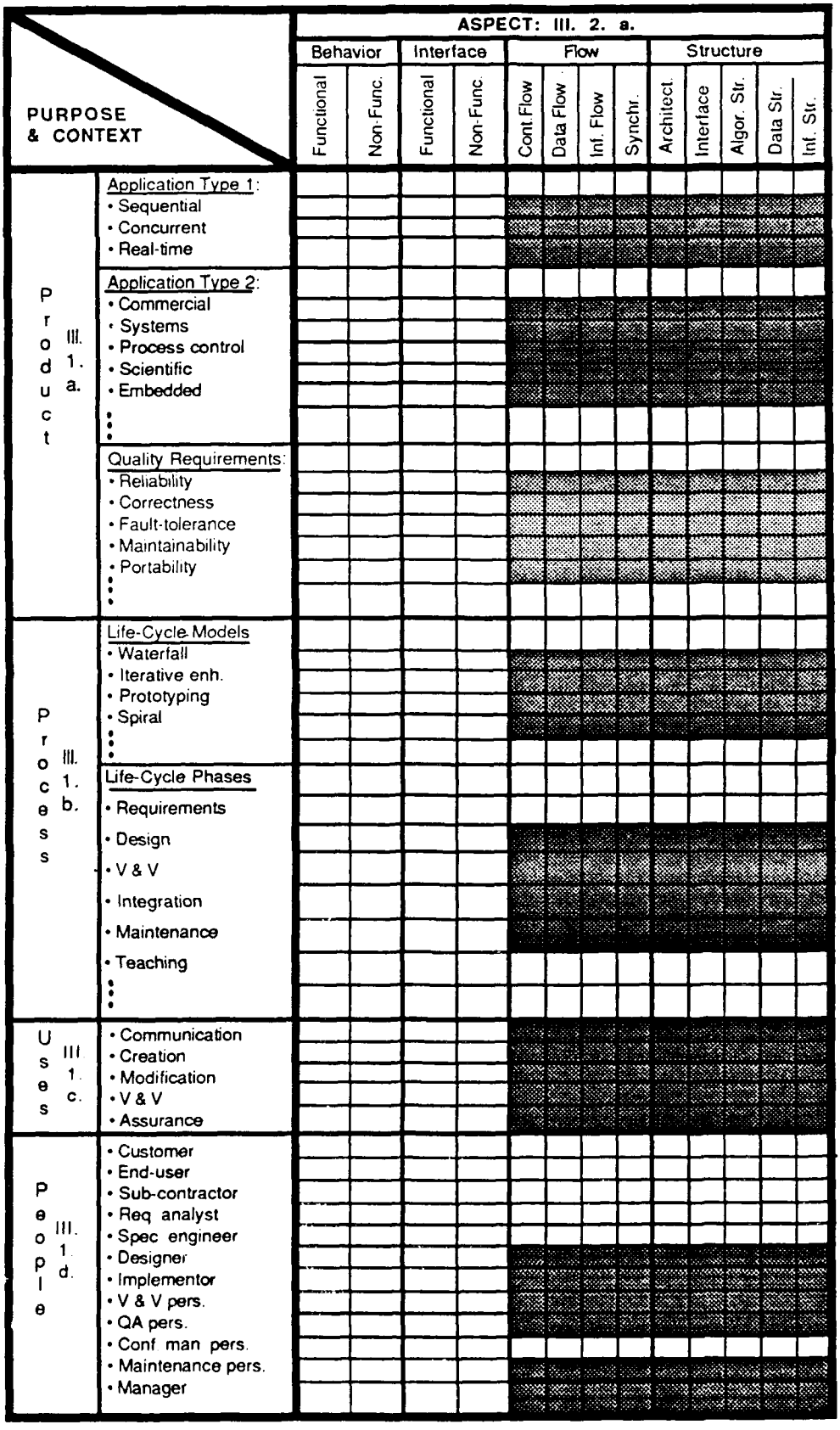


Figure 6. Design Characterization.



Teaching Considerations

Uses of this Material

The material presented in this module is intended to be used in one of three ways:

1. As background material for teachers preparing software engineering courses.
2. As the basis of an introductory unit on requirements (C- or D-requirements) or design.
3. As the basis of a stand-alone course on the selection and assessment of software engineering methods and tools.

Suggested Introductory Literature

The following nine books and papers are recommended as introductory literature on the topics dealt with in this module:

Abbott86	Gomaa86	Lamb88
DeMarco79	Hayes87	Rzepka85
Gehani86	Jensen79	Sommerville89

Suggested Course Schedule

The author has taught the material in this module as a graduate course called "Assessment of Software Requirements Methods and Tools" at the University of Maryland. This course is a stand-alone course on selection and assessment, as suggested above. The planned schedule for this course (14 weeks, 2 hours per week) is shown below. References to the module outline are shown in parentheses.

Week 1	Overview of software evolution (processes, products, etc.).
Week 2	Overview of life-cycle models and the roles played by specifications in these models.
Weeks 3-5	Detailed presentation and discussion of the characterization scheme (III).
Week 6	Presentation and discussion of selection and evaluation criteria for specifications (V). Exercise: An informal software specification is given to four student teams, who are asked to develop corresponding D-requirements documents using any of the following approaches: SADT, Petri nets, R nets, NRL approach, algebraic approach, or axiomatic approach. The teams are asked to justify their choice and to determine the degree to which the method used fulfill their expectations.
Week 7	Presentation and discussion of C-/D-requirements specification (IV.1-2).
Week 8	Presentation and discussion of formal approaches to specifying D-requirements.
Week 9	Other specification types (IV.3-4).
Week 10	Presentation and discussion of exercise by team 1.
Week 11	Presentation and discussion of exercise by team 2.
Week 12	Presentation and discussion of exercise by team 3.
Week 13	Presentation and discussion of exercise by team 4.
Week 14	Course wrap-up.

The requirements document used in the class exercise describes a heating control system. It is one of four informal sets of requirements that have been used as examples within the specification community [IWSSD87].

Exercises

Depending on individual course objectives, the following student exercises may be useful:

1. Distribute an informal requirements document and ask students to create a more formal D-requirements document. (See course description above.) Students can be required either to use a particular method or to chose one themselves from a candidate set of available methods and tools. Students should then assess the effectiveness of the method used.
2. Provide a concrete project scenario (use the characterization scheme in III.1-2) and ask students to chose and justify their choice of the most appropriate specification methods(s) and/or tool(s) (III.3-4).
3. Provide students with corresponding D-requirements, design, and code products. Have them perform modification and/or verification on these products and assess which of the product characteristics are helpful and which cause difficulties in performing the tasks.

Bibliography

Abbott86

Abbott, R. J. *An Integrated Approach to Software Development.* New York: John Wiley, 1986.

This is a general software engineering text, organized as a collection of annotated outlines for product types important to the development and maintenance of software.

Alford77

Alford, M. "A Requirements Engineering Methodology for Real-Time Processing Requirements." *IEEE Trans. Software Eng. SE-3*, 1 (Jan. 1977), 60-69.

Abstract: This paper describes a methodology for the generation of software requirements for large, real-time unmanned weapons systems. It describes what needs to be done, how to evaluate the intermediate products, and how to use automated aids to improve the quality of the product. An example is provided to illustrate the methodology steps and their products and the benefits. The results of some experimental applications are summarized.

Babb85

Babb, R. G., II. "A Data Flow Approach to Unifying Software Specification, Design, and Implementation." *3rd Intl. Workshop on Software Specification and Design.* Washington, D.C.: IEEE Computer Society Press, 1985, 9-13.

Abstract: Specifying requirements for software systems is a complex and frequently frustrating process. A major source of difficulty is that requirements engineering and system development involves a wide range of people, including both computer specialists and non-specialists. This paper describes a unified approach to software specification and design that relies on executable data flow diagrams to serve as a basis for communication among those involved in system development.

Balzer81

Balzer, R., and N. Goldman. "Principles of Good Software Specification and Their Implications for Specification Languages." *AFIPS Conference Proceedings: Vol. 50, 1981 National Computer Conference.* Arlington, Va.: AFIPS Press, 1981, 393-400.

Abstract: Careful consideration of the primary uses of software specifications leads directly to three criteria for judging specifications, which can then be used to develop eight design principles for "good" specifications. These principles, in turn, result in eighteen implications for specification languages that strongly constrain the set of adequate specification languages and identify the need for several novel capabilities such as historical and future references, elimination of variables, and result specification.

Basili75

Basili, V. R., and A. J. Turner. "Iterative Enhancement: A Practical Technique for Software Development." *IEEE Trans. Software Eng. SE-1*, 4 (April 1975), 390-396.

Abstract: This paper recommends the "iterative enhancement" technique as a practical means of using a top-down, stepwise refinement approach to software development. This technique begins with a simple initial implementation of a properly chosen (skeletal) subproject which is followed by the gradual enhancement of successive implementations in order to build the full implementation. The development and quantitative analysis of a production compiler for the language SIMPL-T is used to demonstrate that the application of iterative enhancement to software development is practical and efficient, encourages the generation of an easily modifiable product, and facilitates reliability.

Basili88

Basili, V. R., and H. D. Rombach. "The TAME Project: Towards Improvement-Oriented Software Environments." *IEEE Trans. Software Eng. SE-14*, 6 (June 1988), 758-773.

Abstract: Experience from a dozen years of analyzing software engineering processes and products is summarized as a set of software engineering and measurement principles that argue for software engineering process models that integrate sound planning and analysis into the construction process.

In the TAME (Tailoring A Measurement Environment) project at the University of Maryland we have developed such an improvement-oriented software engineering process model that uses the goal/question/metric paradigm to integrate the constructive and analytic aspects of software development. The model provides a mechanism for formalizing the characterization and planning tasks, controlling and improving projects based on quantitative analysis, learning in a deeper and more systematic way about the software process and product, and feeding the appropriate experience back into the current and future projects.

The TAME system is an instantiation of the TAME software engineering process model as an ISEE (Integrated Software Engineering Environment). The first in a series of TAME system prototypes has been developed. An assessment of experience with this first limited prototype is presented including a reassessment of its initial architecture. The long-term goal of this building effort is to develop a better understanding of appropriate ISEE architectures that optimally support the improvement-oriented TAME software engineering process model.

Berztiss87

Berztiss, A. *Formal Specification of Software.* Curriculum Module SEI-CM-8-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

Capsule Description: *This module introduces methods for the formal specification of programs and large software systems, and reviews the domains of application of these methods. Its emphasis is on the functional properties of software. It does not deal with the specification of programming languages, the specification of user-computer interfaces, or the verification of programs. Neither does it attempt to cover the specification of distributed systems.*

Bjørner82

Bjørner, D., and C. B. Jones. *Formal Specification and Software Development.* Englewood Cliffs, N.J.: Prentice/Hall International, 1982.

Boehm84

Boehm, B. W., T. E. Gray, and T. Seewaldt. "Prototyping vs. Specifying: A Multi-Project Experiment." *Proc. 7th Intl. Conf. Software Eng.* New York: IEEE, 1984, 473-484.

Abstract: *In this experiment, seven software teams developed versions of the same small-size (2000-4000 source instruction) application software product. Four teams used the Specifying approach. Three teams used the Prototyping approach.*

The main results of the experiment were:

Prototyping yielded products with roughly equivalent performance, but with about 40% less code and 45% less effort.

The prototyped products rated somewhat lower on functionality and robustness, but higher on ease of use and ease of learning.

Specifying produced more coherent designs and software that was easier to integrate.

The paper presents the experimental data supporting these and a number of additional conclusions.

Boehm86

Boehm, B. W. "A Spiral Model of Software Development and Enhancement." *ACM Software Engineering Notes 11*, 4 (Aug. 1986), 14-24.

This paper, reprinted from the proceedings of the March 1985 International Workshop on the Software Process and Software Environments, presents Boehm's spiral model. The author's description from the introduction:

> The spiral model of software development and enhancement presented here provides a new framework for guiding the software process. Its major distinguishing feature is that it creates a *risk-driven* approach to the software process, rather than a strictly specification-driven or prototype-driven process. It incorporates many of the strengths of other models, while resolving many of their difficulties.

Brackett90

Brackett, J. W. *Software Requirements.* Curriculum Module SEI-CM-19-1.2, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Jan. 1990.

Capsule Description: *This curriculum module is concerned with the definition of software requirements—the software engineering process of determining what is to be produced—and the products generated in that definition. The process involves all of the following:*

- *requirements identification*
- *requirements analysis*
- *requirements representation*
- *requirements communication*
- *development of acceptance criteria and procedures*

The outcome of requirements definition is a precursor of software design.

Brown87

Brown, B. J. *Assurance of Software Quality.* Curriculum Module SEI-CM-7-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., July 1987.

Capsule Description: *This module presents the underlying philosophy and associated principles and practices related to the assurance of software quality. It includes a description of the assurance activities associated with the phases of the software development life-cycle (e.g., requirements, design, test, etc.).*

Bruno86
Bruno, G., and G. Marchetto. "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems." *IEEE Trans. Software Eng. SE-12*, 2 (Feb. 1986), 346-357.

Abstract: This paper presents a methodology for the rapid prototyping of process control systems, which is based on an original extension to classical Petri nets. The proposed nets, called PROT nets, provide a suitable framework to support the following activities: building an operational specification model; evaluation, simulation, and validation of the model; automatic translation into program structures.

In particular, PROT nets are shown to be translatable into Ada® program structures concerning concurrent processes and their synchronizations. The paper illustrates this translation in detail using, as a worked example, the problem of tool handling in a flexible manufacturing system.

Budgen89
Budgen, D. *Introduction to Software Design.* Curriculum Module SEI-CM-2-2.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Jan. 1989.

Capsule Description: This curriculum module provides an introduction to the principles and concepts relevant to the design of large programs and systems. It examines the role and context of the design activity as a form of problem-solving process, describes how this is supported by current design methods, and considers the strategies, strengths, limitations, and main domains of application of these methods.

Cameron89
Cameron, J. R. *JSP and JSD: The Jackson Approach to Software Development, 2nd Ed.* Washington, D.C.: IEEE Computer Society Press, 1989.

A collection of articles and papers describing JSP and JSD and illustrating these methods using a range of examples of reasonable size and complexity.

Good source material for the instructor. Source of material for student tutorials.

Cohen86
Cohen, B., W. T. Harwood, and M. I. Jackson. *The Specification of Complex Systems.* Reading, Mass.: Addison-Wesley, 1986.

Collofello88
Collofello, J. S. *Introduction to Software Verification and Validation.* Curriculum Module SEI-CM-13-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.

Capsule Description: Software verification and validation techniques are introduced and their applicability discussed. Approaches to integrating these techniques into comprehensive verification and validation plans are also addressed. This curriculum module provides an overview needed to understand in-depth curriculum modules in the verification and validation area.

DeMarco79
DeMarco, T. *Structured Analysis and System Specification.* Englewood Cliffs, N.J.: Yourdon Press, 1979. Also published by Prentice-Hall, 1979.

A very readable book on Structured Analysis and system specification that covers data flow diagrams, data dictionaries, and process specification.

DeRemer76
DeRemer, F., and H. H. Kron. "Programming-in-the-Large Versus Programming-in-the-Small." *IEEE Trans. Software Eng. SE-2*, 6 (June 1976), 80-86.

Abstract: We distinguish the activity of writing large programs from that of writing small ones. By large programs we mean systems consisting of many small programs (modules), usually written by different people.

We need languages for programming-in-the-small, i.e., languages not unlike the common programming languages of today, for writing modules. We also need a "module interconnection language" for knitting those modules together into an integrated whole and for providing an overview that formally records the intent of the programmer(s) and that can be checked for consistency by a compiler.

DoD88a
DoD. *Military Standard for Defense System Software Development.* DOD-STD-2167A, U.S. Department of Defense, Washington, D.C., 29 February 1988.

DoD88b
DoD. *Military Standard for Defense System Software Quality Program.* DOD-STD-2168, U.S. Department of Defense, Washington, D.C., 29 April 1988.

Firth87
Firth, R., *et al. A Classification Scheme for Software Development Methods.* Technical Report CMU/SEI-87-TR-41, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., June 1987.

Abstract: Software development methods are used to assist with the process of designing software for real-time systems. Many such methods have come into practice over the last decade, and new methods are emerging. These new methods are more powerful than the old ones, especially with regard to real-time aspects of the software. This report describes a classification scheme for software development methods, includes descriptions of the major characteristics of such methods, and contains some more words of advice on choosing and applying such methods.

Gehani86
Gehani, N., and A. D. McGettrick, eds. *Software Specification Techniques.* Reading, Mass.: Addison-Wesley, 1986.

A collection of papers on formal specification techniques. This book addresses general principles, particular specification techniques, case studies of actual experiences, and systems for automatic generation of prototypes from specifications.

Goldsack85
Goldsack, S. J. *Ada for Specification: Possibilities and Limitations.* Cambridge, England: Cambridge University Press, 1985.

Gomaa86
Gomaa, H. "Software Development of Real-Time Systems." *Comm. ACM 29*, 7 (July 1986), 657-668.

Suitable for use by both instructors and students as an easily readable introduction to issues of real-time products.

Guttag78
Guttag, J. V., E. Horowitz, and D. R. Musser. "Abstract Data Types and Software Validation." *Comm. ACM 21*, 12 (Dec. 1978), 1048-1064.

Abstract: A data abstraction can be naturally specified using algebraic axioms. The virtue of these axioms is that they permit a representation-independent formal specification of a data type. An example is given which shows how to employ algebraic axioms at successive levels of implementation. The major thrust of the paper is twofold. First, it is shown how the use of algebraic axiomatizations can simplify the process of proving the correctness of an implementation of an abstract data type. Second, semi-automatic tools are described which can be used both to automate such proofs of correctness and to derive an immediate implementation from the axioms. This implementation allows for limited testing of programs at design time, before a conventional implementation is accomplished.

Harel88a
Harel, D. "On Visual Formalisms." *Comm. ACM 31*, 5 (May 1988), 514-530.

An elegant and clearly-written paper that discusses a number of important issues about model representation. While the first part of the paper is concerned with general issues, the latter part provides an interesting exposition of statecharts, and includes a detailed example in the form of a description of a digital watch. The paper will be of particular interest to instructors concerned with the imprecision of the graphical notations frequently used to describe software requirements.

Harel88b
Harel, D., *et al.* "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *Proc. 10th Intl. Conf. Software Eng.* Washington, D.C.: IEEE Computer Society Press, 1988, 396-406.

Abstract: This paper provides a brief overview of the STATEMATE system, constructed over the past three years by i-Logix Inc., and Ad Cad Ltd. STATEMATE is a graphical working environment, intended for the specification, analysis, design and documentation of large and complex reactive systems, such as real-time embedded systems, control and communication systems, and interactive software. It enables a user to prepare, analyze and debug diagrammatic, yet precise, descriptions of the system under development from three inter-related points of view, capturing structure, functionality *and* behavior. *These views are represented by three graphical languages, the most intricate of which is the language of* statecharts *used to depict reactive behavior over time. In addition to the use of statecharts, the main novelty of STATEMATE is in the fact that it 'understands' the entire descriptions perfectly, to the point of being able to analyze them for crucial dynamic properties, to carry out rigorous animated executions and simulations of the described system, and to create runing code automatically. These features are invaluable when it comes to the quality and reliability of the final outcome.*

Hatley87
Hatley, D. J., and I. A. Pirbhai. *Strategies for Real-Time System Specification.* New York: Dorset House, 1987.

This is a well-written text on Real-Time Structured Analysis. This book should be read in conjunction with [Ward89] in order better to understand the capabilities of the notation. This text and [Ward85] are alternative texts; the choice of a text for teaching Real-Time Structured Analysis may depend upon whether the computer tools to be used support only the Hatley notation or only the Ward notation.

Hayes87

Hayes, Ian, ed. *Specification Case Studies.* Englewood Cliffs, N.J.: Prentice/Hall International, 1987.

A collected set of case studies based on the use of Z, providing a well-structured introduction to the use of formal methods. The section on specification of the UNIX filing system may involve sufficiently familiar material to provide a good introduction for many students.

Suitable for use by both instructors and students.

Heninger80

Heninger, K. L. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications." *IEEE Trans. Software Eng. SE-6,* 1 (January 1980), 2-13.

Abstract: This paper concerns new techniques for making requirements specifications precise, concise, unambiguous, and easy to check for completeness and consistency. The techniques are well-suited for complex real-time software systems; they were developed to document the requirements of existing flight software for the Navy's A-7 aircraft. The paper outlines the information that belongs in a requirements document and discusses the objectives behind the techniques. Each technique is described and illustrated with examples from the A-7 document. The purpose of the paper is to introduce the A-7 document as a model of a disciplined approach to requirements specification; the document is available to anyone who wishes to see a fully worked out example of the approach.

Henry81

Henry, S., and D. Kafura. "Software Structure Metrics Based on Information Flow." *IEEE Trans. Software Eng. SE-7,* 5 (Sept. 1981), 510-518.

Abstract: Structured design methodologies provide a disciplined and organized guide to the construction of software systems. However, while the methodology structures and documents the points at which design decisions are made, it does not provide a specific, quantitative basis for making these decisions. Typically, the designers' only guidelines are qualitative, perhaps even vague, principles such as "functionality," "data transparency," or "clarity." This paper, like several recent publications, defines and validates a set of software metrics which are appropriate for evaluating the structure of large-scale systems. These metrics are based on the measurement of information flow between system components. Specific metrics are defined for procedure complexity, module complexity, and module coupling. The validation, using the source code for the UNIX operating system, shows that the complexity measures are strongly correlated with the occurrence of changes. Further, the metrics for procedures and modules can be interpreted to reveal various types of structural flaws in the design and implementation.

Hoare69

Hoare, C. A. R. "An Axiomatic Basis for Computer Programming." *Comm. ACM 12,* 10 (Oct. 1969), 576-580.

Abstract: In this paper an attempt is made to explore the logical foundation of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

IEEE83

IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* New York: IEEE, 1983. ANSI/IEEE Std 729-1983.

Provides definitions for many of the terms used in software engineering.

IEEE84

IEEE. *IEEE Guide to Software Requirements Specifications.* New York: IEEE, 1984. ANSI/IEEE Std 830-1984.

IWSSD82

First Intl. Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1982.

IWSSD84

2nd Intl. Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1984.

IWSSD85
3rd Intl. Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985.

IWSSD87
4th Intl. Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1987. Also appears as *ACM Software Engineering Notes 14*, 3 (May 1989).

Jensen79
Jensen, R. W., and C. C. Tonies. *Software Engineering.* Englewood Cliffs, N.J.: Prentice-Hall, 1979.

A collection of primarily management-oriented articles. Structured program design is covered.

Lamb88
Lamb, David Alex. *Software Engineering: Planning for Change.* Englewood Cliffs, N.J.: Prentice-Hall, 1988.

This book introduces basic software engineering concepts. Among other topics, it contains an elaborate discussion of "specification and verification." Specific emphasis is placed on algebraic specifications, trace specifications, and abstract modeling.

Levine89
Levine, Linda, Linda H. Pesante, and Susan B. Dunkle. *Technical Writing for Software Engineers.* Curriculum Module SEI-CM-23-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1989.

Capsule Description: *This module, which is directed specifically to software engineers, discusses the writing process in the context of software engineering. Its focus is on the basic problem-solving activities that underlie effective writing, many of which are similar to those underlying software development. The module draws on related work in a number of disciplines, including rhetorical theory, discourse analysis, linguistics, and document design. It suggests techniques for becoming an effective writer and offers criteria for evaluating writing.*

Mills86
Mills, H. D., C. Linger, and A. R. Hevner. *Principles of Information Systems Analysis and Design.* Orlando, Fla.: Academic Press, 1986.

This book describes an approach to requirements definition for information systems that emphasizes the use of models showing external system behavior. Black-box and state-machine models are used, which are similar in concept to the form of representation described in [Heninger80].

Parnas72
Parnas, D. L. "On the Criteria to be used in decomposing systems into modules." *Comm. ACM 15*, 12 (Dec. 1972), 1053-1058.

Abstract: *This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.*

A truly "classical" paper, in the sense of being often cited but probably rarely read. It is a very important paper that lays down the basic ideas about information hiding but in a very concise and compact form. The discussion is based upon an example of a problem that may not be very familiar to many readers.

The teacher must read this paper; the student might do better to settle for the teacher's interpretation.

Peterson77
Peterson, J. "Petri Nets." *ACM Computing Surveys 9*, 3 (Sept. 1977), 223-252.

This is the first widely circulated survey and tutorial on Petri nets. It touches briefly on modeling with Petri nets, basic definitions, analysis problems and techniques, Petri net languages, and related models of computation. A good introduction that should be readable by any graduate student.

Peterson81
Peterson, J. L. *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, N.J.: Prentice-Hall, 1981.

This books makes two important contributions: it identifies a new class (called Petri net languages) in the Chomsky hierarchy, and it organizes a set of models of parallel computation into a lattice in which the ordering is based on the expressive power of a model. Examples are given to show the proper

inclusion among each adjacent pair of models in the lattice. An excellent bibliography is provided.

Ross77

Ross, D. T., and K. E. Schoman, Jr. "Structured Analysis for Requirements Definition." *IEEE Trans. Software Eng. SE-3*, 1 (Jan. 1977), 6-15.

Abstract: Requirements definition encompasses all aspects of system development prior to actual system design. We see the lack of an adequate approach to requirements definition as the source of major difficulties in current systems work. This paper examines the needs for requirements definition, and proposes meeting those objectives with three interrelated subjects: context analysis, functional specification, and design constraints. Requirements definition replaces the widely used, but never well-defined, term "requirements analysis."

The purpose of this paper is to present, in a comprehensive manner, concepts that apply throughout requirements definition (and, by implication, to all of system development). The paper discusses the functional architecture of systems, the characteristics of good requirements documentation, the personnel involved in the process of analysis, and management guidelines that are effective even in complex environments.

The paper then outlines a systematic methodology that incorporates, in both notation and technique, the concepts previously introduced. Reference is made to actual requirements definition experience and to practicable automated support tools that may be used with the methodology.

Royce70

Royce, W. W. "Managing the Development of Large Software Systems: Concepts and Techniques." *WESCON Technical Papers Volume 14, Western Electronic Show and Convention.* Los Angeles: WESCON, 1970, 1-9. Reprinted in *Proc. 9th Intl. Conf. Software Eng.*, Washington, D.C.: IEEE Computer Society Press, 1987, 328-338.

Abstract: Gives the personal views of the author about managing large software developments. He has had various assignments during the past nine years, mostly concerned with the development of software packages for spacecraft mission planning, commanding and post-flight analysis. In these assignments he has experienced different degrees of success with respect to arriving at an operational state, on-time, and within costs. He has become prejudiced by his experiences and relates some of these prejudices in the presentation.

Rzepka85

Special Issue on Requirements Engineering Environments. W. Rzepka and Y. Ohno, eds. *Computer 18*, 4 (April 1985).

The papers in this issue cover approaches such as SADT and SREM, with special emphasis on real-time applications.

Scacchi87

Scacchi, W. *Models of Software Evolution: Life Cycle and Process.* Curriculum Module SEI-CM-10-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

Capsule Description: This module presents an introduction to models of software system evolution and their role in structuring software development. It includes a review of traditional software life-cycle models as well as software process models that have been recently proposed. It identifies three kinds of alternative models of software evolution that focus attention to either the products, production processes, or production settings as the major source of influence. It examines how different software engineering tools and techniques can support life-cycle or process approaches. It also identifies techniques for evaluating the practical utility of a given model of software evolution for development projects in different kinds of organizational settings.

Sommerville89

Sommerville, I. *Software Engineering, 3rd Ed.* Wokingham, England: Addison-Wesley, 1989.

This book contains an easy-to-read introduction to software engineering principles and issues. It emphasizes the early life-cycle stages, including "software specification."

Sutcliffe88

Sutcliffe, A. *Jackson System Development.* New York: Prentice-Hall, 1988.

From the introductory chapter:

> [Jackson System Development (JSD)] is organized in three separate stages which guide the analyst through the systems development process. Each stage has a set of activities with clear start and end points (this helps the analyst using the method) and facilitates project control as deliverables can be defined for each stage. The three stages can be outlined briefly as follows.
>
> (a) *Modelling stage*. A description is made of the real world problem and the important actions within the system are identified. This is followed by analysis of the major structures within the system, called *entities* in JSD. . . .
>
> (b) *Network stage*. The system is developed

> as a series of subsystems. First the major structures are taken from the modelling stage and input and outputs are added; this is followed by the analysis of the output subsystem which provides information, and then of the input subsystem which handles the user interface and validation. . . .
>
> (c) *Implementation stage.* In this stage the logical system specification, which is viewed as a network of concurrently communicating processes, is transformed into a sequential design by the technique of scheduling. This is followed by further detailed design and coding. . . .
>
> JSD begins by analysing the major system structures which are important to create a model of the system problem, the entities. Then these structures are connected together to create a network model of the system, while at the same time the design is elaborated by addition of other processes to create output, and to handle input messages and user interaction. The essence . . . is to create a system model of reality first and then to add the functionality.

JSD is usually not considered to support requirements definition, but Jackson's emphasis on modeling the problem domain makes it a viable alternative, for information systems, to functional, top-down approaches such as Structured Analysis. This book is unique in showing how JSD relates to more widely used software requirements and design techniques. [Ward89] also shows how its notation relates to more widely used requirements notations.

Teichrow77

Teichrow, D. "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems." *IEEE Trans. Software Eng. SE-3*, 1 (Jan. 1977), 41-48.

> ***Abstract:*** *PSL/PSA is a computer-aided structured documentation and analysis technique that was developed for, and is being used for, analysis and documentation of requirements and preparation of functional specifications for information processing systems. The present status of requirements definition is outlined as the basis for describing the problem which PSL/PSA is intended to solve. The basic concepts of the Problem Statement Language are introduced and the content and use of a number of standard reports that can be produced by the Problem Statement Analyzer are briefly described.*
>
> *The experience to date indicates that computer-aided methods can be used to aid system development during the requirements definition stage and that the main factors holding back such use are not so much related to the particular characteristics and capabilities of PSL/PSA as they are to organizational considerations involved in any change in methodology and procedure.*

Tomayko87

Tomayko, J. E. *Software Configuration Management.* Curriculum Module SEI-CM-4-1.3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., July 1987.

> ***Capsule Description:*** *Software configuration management encompasses the disciplines and techniques of initiating, evaluating, and controlling change to software products during and after the development process. It emphasizes the importance of configuration control in managing software production.*

Ward85

Ward, P. T., and S. J. Mellor. *Structured Development for Real-Time Systems.* New York: Yourdon Press, 1985-1986. The three volumes in this series are *Introduction and Tools*, *Essential Modeling Techniques*, and *Implementation Modeling Techniques.*

> This book is an alternative to [Hatley87] for teaching Real-Time Structured Analysis.

Ward89

Ward, P. T. "Embedded Behavior Pattern Languages: A Contribution to a Taxonomy of CASE Languages." *J. Syst. and Software 9*, 2 (Feb. 1989), 109-128.

> ***Abstract:*** *With the increasing availability of CASE tools, graphics-based software modeling languages have the potential to play a much more central role in the development process. Although some comparisons among these languages have been made, no systematic classification based on the underlying abstractions has been attempted. As a contribution to such a classification, a class of languages designated Embedded Behavior Pattern (EBP) languages is described and its members are compared and contrasted. The EBP languages include the Ward/Mellor and Boeing/Hatley Structured Analysis extensions, the Jackson System Development notation, and Harel's StateChart-Activity Chart notation. These notations are relevant to the building of specification models because they display clear one-to-one correspondences between elements of the model and elements of the application domain. These notations are also amenable to a style of model partitioning that is related to object-oriented development.*
>
> This paper is a detailed comparison of the notations described in [Harel88a], [Hatley87], and [Ward85].

Webster87

Webster's Ninth New Collegiate Dictionary. Springfield, Mass.: Merriam-Webster, 1987.

Yourdon89
Yourdon, E. *Modern Structured Analysis.* Englewood Cliffs, N.J.: Yourdon Press, 1989.

Probably the most comprehensive and up-to-date book on the popular Structured Analysis method.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SEI-CM-11-2.0	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. / 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/AVS HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE / 8b. OFFICE SYMBOL (If applicable) ESD/ AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: 63752F / PROJECT NO.: N/A / TASK NO.: N/A / WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) Software Specifications: A Framework	
12. PERSONAL AUTHOR(S) H. Dieter Rombach, University of Maryland	
13a. TYPE OF REPORT FINAL / 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) December 1989 / 15. PAGE COUNT 36
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) specification requirements specification specification document

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This curriculum module presents a framework for understanding software product and process specifications. An unusual approach has been chosen in order to be able to address all aspects related to "specification" without confusing the many existing uses of the term. In this module, the term specification refers to any plan (or standard) according to which products of some type are constructed or processes of some type are performed, not to the products or processes themselves. In this sense, a specification is itself a product that describes how products of some type should be performed. The framework includes a reference software life-cycle model and terminology; a characterizing scheme for software product and process specifications; guidelines for using the characterization scheme to identify clearly certain lifecycle phases; and guidelines for using the characterization scheme to select and evaluate specification techniques.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION
22a. NAME OF RESPONSIBLE INDIVIDUAL JOHN S. HERMAN, Capt, USAF	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630 / 22c. OFFICE SYMBOL SEI JPO

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Software Engineering Curriculum Project is developing a wide range of materials to support software engineering education. A *curriculum module* (CM) identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in designing a course. A *support materials* package (SM) contains materials related to a module that may be helpful in teaching a course. An *educational materials* package (EM) contains other materials not necessarily related to a curriculum module. Other publications include software engineering curriculum recommendations and course designs.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

Comments on SEI educational materials and requests for additional information should be addressed to the Software Engineering Curriculum Project, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213. Electronic mail can be sent to education@sei.cmu.edu on the Internet.

Curriculum Modules (* Support Materials available)

CM-1 [superseded by CM-19]
CM-2 Introduction to Software Design
CM-3 The Software Technical Review Process*
CM-4 Software Configuration Management*
CM-5 Information Protection
CM-6 Software Safety
CM-7 Assurance of Software Quality
CM-8 Formal Specification of Software*
CM-9 Unit Testing and Analysis
CM-10 Models of Software Evolution: Life Cycle and Process
CM-11 Software Specifications: A Framework
CM-12 Software Metrics
CM-13 Introduction to Software Verification and Validation
CM-14 Intellectual Property Protection for Software
CM-15 Software Development and Licensing Contracts
CM-16 Software Development Using VDM
CM-17 User Interface Development*
CM-18 [superseded by CM-23]
CM-19 Software Requirements
CM-20 Formal Verification of Programs
CM-21 Software Project Management
CM-22 Software Design Methods for Real-Time Systems*
CM-23 Technical Writing for Software Engineers
CM-24 Concepts of Concurrent Programming
CM-25 Language and System Support for Concurrent Programming*
CM-26 Understanding Program Dependencies

Educational Materials

EM-1 Software Maintenance Exercises for a Software Engineering Project Course
EM-2 APSE Interactive Monitor: An Artifact for Software Engineering Education
EM-3 Reading Computer Programs: Instructor's Guide and Exercises